Planck

A Domain Specific Language for Programming Haptic Devices

by

Victor I. Barua

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE

in

The Faculty of Science

(Combined Honours Computer Science and Physics)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

May 2015

Abstract

The Planck language provides users with high-level mathematical abstractions with which to encode mathematical and/or physical models. With a succinct description of a model, Planck programs can generate interactive computer-based simulations with basic graphical output. Planck programs can also be used to generate code that runs on hardware devices, allowing users to physically interact with their models.

This thesis motivates the creation of the Planck language in the context of haptics based education for physics and mathematics. It details the proof-of-concept implementation of the language and provides usage examples to help demonstrate the capabilities of the language.

Table of Contents

Abstract . . . iii

Table of Contents . . . v

Acknowledgements . . . ix

1 Introduction . . . 1

2 Background . . . 3
2.1 Hapkit . . . 3
2.2 Numerical Methods . . . 3

3 A Practical Introduction to Planck . . . 7
3.1 Design Goals . . . 7
3.2 Basics . . . 7
3.2.1 Numerical Expressions . . . 8
3.2.2 Logical Expressions . . . 8
3.2.3 Conditional Branching . . . 8
3.2.4 Definitions . . . 8
3.2.5 Special Identifiers . . . 9
3.2.6 Integration and Differentiation . . . 9
3.2.7 Rendering Expressions . . . 9
3.3 Interpreted Programs . . . 9
3.3.1 Simplest Program . . . 9
3.3.2 Stationary Ball . . . 10
3.3.3 Spring . . . 10
3.3.4 Keyboard Input . . . 11
3.3.5 Spring with Keyboard Input . . . 11
3.4 Compiled Programs . . . 12
3.4.1 Simplest Program . . . 12
3.4.2 Spring . . . 13
3.5 Running Planck Programs . . . 13

4 A Formal Introduction to Plank . . . 15
4.1 Overview . . . 15
4.2 Constraints . . . 15
4.3 Influences . . . 15
4.4 Language Representation . . . 16
4.4.1 Programs - PROGRAM . . . 16

4.4.2 Definitions - DEFN . . . 17
4.4.3 Expressions - EXPR . . . 17
4.4.4 Rendering - IMAGE . . . 19

5 Preprocessing . . . 21
5.1 Desugaring . . . 21
5.1.1 Conditional Desugaring . . . 21
5.1.2 Integration and Derivative Desugaring . . . 22
5.2 Identifier Injection . . . 22
5.3 Preprocessing an Interpreted Program . . . 23
5.4 Preprocessing a Compiled Program . . . 25

6 Interpretation . . . 27
6.1 Interpretation Strategy . . . 27
6.1.1 Initialization Phase . . . 28
6.1.2 Render Phase . . . 29
6.1.3 Input Phase . . . 29
6.1.4 Update Phase . . . 29
6.1.5 Example . . . 30

7 Compilation . . . 35
7.1 Templates . . . 35
7.1.1 Imports . . . 35
7.1.2 Definitions . . . 35
7.1.3 Time . . . 37
7.1.4 World . . . 37
7.1.5 Integration and Differentiation . . . 37
7.1.6 Expressions . . . 37
7.1.7 Protoypes . . . 39
7.1.8 State Initialization and State Functions . . . 39
7.1.9 Device Specific Code . . . 39
7.1.10 setup . . . 40
7.1.11 loop . . . 42

8 Conclusion . . . 43

Bibliography . . . 45

Appendices . . .

A Extended BNF . . . 47

B Language.rkt . . . 49

C Parser.rkt . . . 53

D Desugar.rkt . . . 57

E ExpressionStore.rkt 63

F World.rkt 65

G Preprocessing.rkt 67

H Interpretation.rkt 69

I CodeGeneration.rkt 75

J Planck.rkt 83

K General Template Format 85

L Hapkit Template 89

M Compiled Spring Program 95

Acknowledgements

Throughout my thesis I had the pleasure of working closely with two fine individuals.

Thank you Dr. Steven Wolfman for introducing me to the study of programming languages, and for poking lots of holes in my schemes to make them stronger.

Thank you Dr. Ronald Garcia for guiding me further into the wonderful world of programming languages, and providing honest insight into the grand life of research.

Both of you have been amazing supervisors. Supportive and questioning, I could not have asked for more.

Finally, thank you to my family for supporting me in all of my endeavors, even when they have kept me far from home.

Chapter 1

Introduction

A key learning objective in physics courses is the ability of students to visualize and understand the consequences of the equations that are developed during its study. Knowing that acceleration is the second derivative of position with respect to time does not inherently enable individuals to predict how a physical system will behave. Developing such an intuition is an important part of studying physics, and finding ways to develop this intuition in students an important aspect of teaching physics.

One of the oldest ways of developing such intuitions is by experimenting with actual physical systems. Pulling on a spring gives clear tactile feedback of the force it generates and can aid in understanding the equations that govern it. However not all physics can be explored easily or safely with physical experiments. In such cases though it may still be possible to explore the physics involved through simulated visualizations, like those provided by the PhET Interactive Simulations [6] project. An important aspect of these visualizations is the ability for students to tweak the parameters of the simulation and observe the results immediately. The ease of changing physical parameters on the fly is one advantage that simulated systems have over physical systems. On the other hand, simulated systems would be hard pressed to provide tactile feedback

The Hapkit [8] project aims to bridge this gap. It is one of the projects in the Cyberlearning and Future Learning Technologies program, run by the National Science Foundation, which aims to develop and explore new technologies for use in classrooms [4]. The Hapkit itself is a haptic paddle controlled by an Arduino [1] with the ability to both sense its current orientation and apply force to the paddle. Simulations running on a Hapkit can provide both visual feedback (by the motion of the paddle) and tactile feedback (by touching the paddle). One goal of the Hapkit project is to allow students and educators to program and explore their own virtual environments.

Setting up a virtual environment in the Hapkit requires an understanding of the I/O mechanism of the Arduino board, the ability to set up a robust feedback loop mechanism and mapping of virtual inputs and outputs to real inputs and outputs. For end users like students and teachers seeking to explore physic and mathematics with the Hapkit, this is well beyond the scope of what they are expected to know. The Planck domain specific language (DSL) aims to makes this as simple as possible by allowing these users to encode their simulations using mathematical operators that they should be familiar with from their study of physics. From this high-level mathematical description, Planck programs can be either be interpreted to generate computer based visualizations or compiled to Arduino C to run on a Hapkit.

Chapter 2

Background

The following section provides background information on the Hapkit device, which is the proof-of-concept hardware target of Planck. It also provides information on the numerical methods used to run simulations within Planck.

2.1 Hapkit

The field of haptics is concerned with methods of providing tactile sensory feedback to individuals. It combines knowledge of how humans perceive tactile sensations with mechatronics in order to build devices that are capable of providing realistic feedback. Applications of haptics range from the simple non-interactive silent vibration mode available in most cellular devices to systems that allow surgeons to perform procedures remotely by providing high fidelity feedback. Systems providing feedback are normally reactive systems that have both sensory capabilities and some way of outputting a physical response. In order to achieve high fidelity these systems normally require a tight control loop, which in practice corresponds to update frequencies on the order of 1 kHz.

In an effort to expand the teaching of haptics, the Hapkit project aims to produce a cheap and easy to assemble haptic paddle. Figure 2.1 shows a labelled diagram of a Hapkit. Haptic paddles have three main components: a paddle, a motor to drive the paddle and a sensor to determine the orientation of the paddle. In the Hapkit the motor is physically coupled to the paddle using a drive wheel. As the motor rotates, the drive wheel rotates and the friction between the drive wheel and the paddle causes the paddle to turn. A magnetoresistor sensor, once calibrated, can be used to count the rotations of the drive wheel and correlate those to the position of the paddle handle. For additional information on the Hapkit, there is a free online course [10] on haptics that uses the Hapkit.

With the Hapkit, an end user could program haptic virtual environments and interact with them via the paddle. The implementation and simulation of these virtual environments lies at the core of the Planck project.

2.2 Numerical Methods

Numerical differentiation and integration are vital for many physics simulations and are a key component of creating realistic virtual environments. Many physical quantities can be related to one another via derivatives. For example Equation 2.1 encodes the relationships between velocity $v(t)$ and position $x(t)$.

$$v(t) = \frac{dx(t)}{dt} \tag{2.1}$$

With this relationship in hand, if $x(t)$ is known then $v(t)$ can be determined by taking the derivative of $x(t)$ with respect to time. If on the other hand $v(t)$ and some initial condition

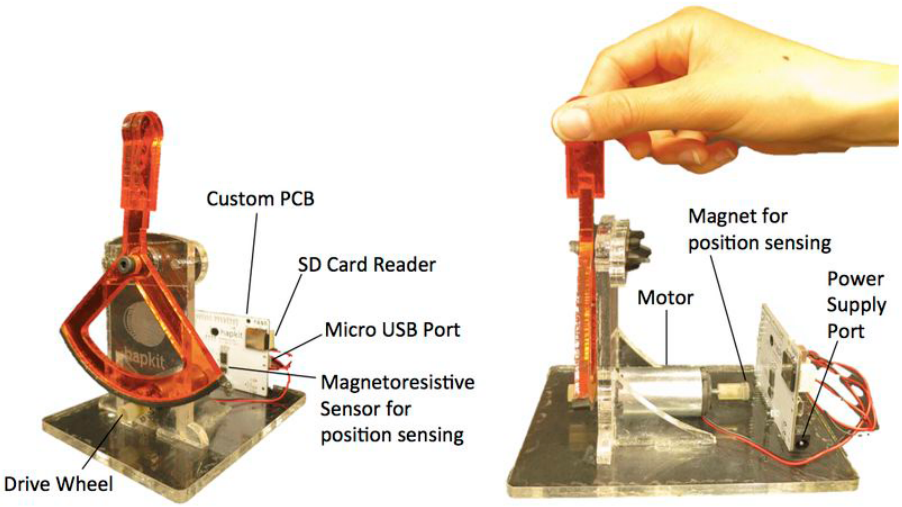

Figure 2.1: Labelled diagram of the Hapkit [9]

$x(0) = x_0$ is known then

$$x = \int_0^t v(s)ds \tag{2.2}$$

However these relationships may not always be as simple. In some cases quantities may be related by ordinary differential equations. An example of this is Hooke's Law for springs, shown in Equation 2.3.

$$F = ma = m\frac{d^2x}{dt^2} = -kx \tag{2.3}$$

Thus when choosing numerical methods for performing general differentiation and integration it is important to choose methods robust enough to deal with both simple and complex relationships.

Given $f(t)$, one of the simplest methods for performing numerical differentiation and determining $f'(t)$ is the Forward Difference method shown in Equation 2.4

$$f'(t_i) = \frac{f(t_i) - f(t_{i-1})}{t_i - t_{i-1}} \tag{2.4}$$

Given $y(t) = f'(t)$ and $y(0) = y_0$, a simple method for numerical integration is the Forward Euler method shown in Equation 2.5

$$y(t_i) = y(t_{i-1}) + (t_i - t_{i-1})f'(t_{i-1}) \tag{2.5}$$

However whilst these methods are simple they both have the disadvantage of having the error at the i^{th} step scale as $O(t_i - t_{i-1})$. Better methods for numerical differentiation and integration exist that improve on this error, however in the context of running the simulations in real time certain constraints arise that make them difficult to implement.

For a pure simulation it is possible to fix the step size between time points such that $t_i - t_{i-1} = h$. In a system running in real time though h need not be fixed. Thus methods that assume a fixed step-size to improve accuracy are no longer applicable. Whilst it is possible to work around this, these workarounds would introduce additional complexity and computational steps into the

numerical algorithms. These additional steps would increase the time to compute a single loop and thus increase the time between time points, decreasing accuracy.

Thus there exists a tradeoff between simple methods which can be computed quickly but have a lower accuracy and more sophisticated methods that take longer to compute but improve accuracy. Investigating this trade-off could be the topic of an entire project itself. In order to complete the proof-of-concept Planck implementation the choice has been made to proceed with the simple methods shown in this section.

Chapter 3

A Practical Introduction to Planck

The following chapter briefly explains the design goals of the Planck language before diving in to explore its capabilities. It explains the difference between interpreted and compiled Planck programs and provides examples of both. A more formal introduction to the language is available in Chapter 4.

3.1 Design Goals

The problem that Planck aims to solve is that of writing programs for Haptic devices that can react to user input. The current iteration of the online Stanford Haptics course [10] provides students with Arduino C programs in which students can tweak parameters to change the behaviour of the Hapkit system. Whilst this is suitable for teaching a course on haptics with the Hapkit, for individuals looking to use the Hapkit to explore mathematical or physical behaviours it can represent a high barrier for entry by requiring them to be familiar with Arduino C as well as Arduino and haptics programming patterns.

The primary design goal governing the development of Planck was to separate the logic that encodes the physical or mathematical model that the user desires to simulate from the implementation of the simulation. This goal is achieved via two main thrusts. First by providing high level mathematics abstractions that allow users to translate equations almost one-to-one into Planck expressions. Second by abstracting away the specifics of where the simulation is being run as much as possible. Both of these thrusts work to minimize the background knowledge a user needs to create a virtual environment simulation.

The second thrust also has a beneficial effect of helping the portability of haptics programs. A compiled Planck program written for the Hapkit could, with some small tweaks, be used on another class of device entirely. This will be further explained in Chapter 7

3.2 Basics

There are two types of Planck programs. Interpreted Planck programs run on a host machine and can generate visualizations from the simulation. Compiled Planck programs are used to generate Arduino C code which can be loaded onto a Hapkit. From here the simulation can run directly on the Hapkit and generate physical outputs.

All Planck programs, regardless of their type, contain a list of expressions bound to identifiers. Using these it is possible to map mathematical and physical equations into Planck expressions in a straightforward manner. The different kinds of expressions available follow.

3.2.1 Numerical Expressions

In addition to the the traditional mathematical operators of +, −, × (as `*`) and /, Planck also supports basic trigonometric operators (sin, cos, tan) as well as squaring of numbers (sqr) and taking the square root of numbers (sqrt).

- `{+ 1 2}`
- `{* 3 4}`
- `{sin 5}`
- `{sqrt 6}`

3.2.2 Logical Expressions

Planck supports numerical comparisons like equality (`=`) as well as inequality comparisons (`<`, `<=`, `>`, `>=`. Combinations of logical values can be achieved via `and` and `or`.

- `{= 1 2}`
- `{and {< 3 4} {<= 5 6}}`

3.2.3 Conditional Branching

`if` expressions allow for conditional branching in Planck programs.

- ```
  {if {< 1 2}
      1
      {- 1}}
  ```

If the first expression in an `if` block evaluates to `True` the value of the second expression is returned, otherwise the value of the third expression is returned. The first expression within an `if` block must evaluate to a boolean value.

3.2.4 Definitions

The binding of an expression to an identifier is referred to as a definition. Definitions are composed of an identifier followed by an expression.

- `{x {+ 1 2}}`
- `{y {= 3 4}}`

Bound identifiers can also be used within expressions.

- `{z {sin x}}`

A program attempting to use an unbound identifier will throw an error. Identifiers can only contain alphanumeric characters and underscores, but may not begin with underscores or numbers. For example, whilst `F_1` is a valid identifier, `1_F` and `_F1` are not.
  ```

3.2.5 Special Identifiers

Interpreted Planck programs have access to a special identifier `t` which evaluates to the current time in the simulation. They also have access to the identifiers `ku`, `kd`, `kl` and `kr` which map to the state of the up, down, left and right arrow keys respectively. If the corresponding key is pressed down the identifier evaluates to 1, otherwise it evaluates to 0.

3.2.6 Integration and Differentiation

Planck supports both numerical integration and differentiation with respect to time through the `integrate` and `derivative` operators.

- `{position {integrate velocity}}`
- `{velocity {derivative position}}`

3.2.7 Rendering Expressions

Rendering expressions are unique to interpreted Planck programs. They cannot be used within definitions (their usage will be shown later on). Only circles can be rendered by the proof-of-concept version of Planck. A `circle` expression is defined as

```
{circle radius color x_pos y_pos}
```

where *radius*, *x_pos* and *y_pos* are either identifiers defined previously or numbers. *color* is a string such as `"red"`, `"green"` or `"blue"`. Both *x_pos* and *y_pos* correspond roughly to pixels on a Cartesian grid.

3.3 Interpreted Programs

An interpreted Planck program is structured as a list of definitions followed by a list of rendering expressions and then by two numbers. The first number is the simulation tick rate which indicates the real time between ticks of the simulation. The second number is the simulation step size which sets the step size used by the numerical methods for integration and differentiation on each tick.

For example with a tick rate of 0.01 and step size of 0.1 the simulation will tick every 0.01 seconds of real time but the simulation time will advance by 0.1 seconds on every tick. In this case the simulation runs "faster" than real time. If both values are the same real time and simulation time will coincide, and if the tick rate is larger than the step size the simulation will run "slower" than real time.

3.3.1 Simplest Program

The simplest possible interpreted program, shown in Figure 3.1, performs no computations but does serve to showcase the general structure of interpreted programs.

In more interesting programs (like those that follow), the first `{}` contains definitions and the second `{}` contains rendering expressions. The simulation tick rate and step size are the same in this example so simulation time flows in step with real time.

```
{{}
 {}
 1
 1}
```

Figure 3.1: The simplest interpreted program possible. It does nothing.

3.3.2 Stationary Ball

The program in Figure 3.2 renders two stationary balls. Note the use of the bound identifier x within the rendering expressions.

```
{{{x 100}}
 {{circle x "red" 0 0}
  {circle 10 "green" x 50}}
  0.01
  0.02}
```

Figure 3.2: Rendering two stationary balls.

In this program the identifier `x` evaluates to the the number 100. The first rendering expression draws a red circle at position $(0, 0)$ with a radius given by the value of `x` (which is 100). The second rendering expression draws a green circle at position $(x, 50) = (100, 50)$ with radius of 10.

The tick rate of this program is 0.01 and the step size is 0.02, so the simulation runs twice as fast as real time.

3.3.3 Spring

An ideal spring is governed by Hooke's Law (Equation 3.1).

$$F = -kx \tag{3.1}$$

When combined with Newton's 2$^{\text{nd}}$ Law (Equation 3.2)

$$F = m\frac{d^2x}{dt^2} \tag{3.2}$$

this gives a simple model of a spring which can be encoded within Planck.

$$m\frac{d^2x}{dt^2} = -kx \tag{3.3}$$

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x = -cx, \qquad c = \frac{k}{m} \tag{3.4}$$

$$x = \int_0^t \int_0^t -cx\, dt^2 \tag{3.5}$$

The relationship in Equation 3.5 maps to the Planck program in Figure 3.3.

This program draws a red circle of radius 20 at position $(x, 0)$. Unlike the previous program `x` does not evaluate to a constant value so the circle will move back an forth along the x-axis.

```
{{{x {+ {integrate {integrate {* {- c} x}}} x0}}
  {x0 400}
  {c 200}}
 {{circle 20 "red" x 0}}
 0.003
 0.001}
```

Figure 3.3: Simulating a spring.

Two numbers `x0` and `c` are defined. The first is an initial offset used to give the spring an initial perturbation (otherwise it would stay at 0 for the duration of the program which would make for an uninteresting simulation). The second corresponds to c as used in Equation 3.5. Both of these numbers were choses to given an aesthetically pleasing simulation (ie. the circle is clearly seen to oscillate back and forth).

The tick rate of this program is 0.003 and the step size is 0.001, so the simulation runs three times slower than real time.

3.3.4 Keyboard Input

Combining keyboard input identifiers with the `integrate` operator allows for the accumulation of user input. In the program in Figure 3.4 the `x` and `y` identifiers map to the coordinates of the rendered ball. The factor `c` is used to increase the effect of depressing one of the keyboard inputs. The subtraction of `kl` from `kr` makes the value of `x` increase when `kr` is depressed and decrease when `kl` is depressed. The same happens for the combination of `ku` and `kd`. The single integration in `x` corresponds to position control along the x-axis, whilst the double integration in `y` corresponds to velocity control along the y-axis.

```
{{{c 1000}
  {x {integrate {* c {- kr kl}}}}
  {y {integrate {integrate {* c {- ku kd}}}}}}
 {{circle 10 "red" x y}}
 0.001
 0.001}}
```

Figure 3.4: An interpreted program using keyboard inputs.

The tick rate and step size of this program are both 0.001 so the simulation runs at the same speed as real time.

3.3.5 Spring with Keyboard Input

The program in Figure 3.5 encodes a spring rendered as a circle whose radius varies based on whether the up arrow is pressed. Note how the red circle is used to visualize the position of the spring along the x-axis, whilst the blue circle is used to visualize the velocity along the y-axis. This program will be used in later sections to demonstrate program preprocessing and interpretation.

```
{{{x  {+ {integrate v} 100}}
  {v  {integrate F}}
  {F  {- {* 200 x}}}
  {R  {+ {* 40 ku} 10}}}
 {{circle R "red" x 0}
  {circle 10 "blue" 0 v}}
 0.001
 0.001}
```

Figure 3.5: An interpreted Planck program simulating a spring which reacts to keyboard input.

3.4 Compiled Programs

The process of compiling Planck programs to Arduino C to run on the Hapkit makes use of a special template file that defines a number of parameters. *xh* holds the current x-position of the paddle. *force* is used to determine how much force the motor should apply. *globalTime* keeps track of the current time in the simulation.

In addition a number of parameters need to be set by the program. *rPulley* (radius of the motor pulley), *rSector* (radius of the sector pulley) and *rHandle* (radius of the handle) can all be measured from the Hapkit itself. m and *theta0* are calibration parameters used to map from a count of rotations from the magnetoresistor to the x-position of the paddle. For additional details on measuring and calibrating the Hapkit see the online course [10] mentioned previously.

The complete structure of a compiled Planck program consists of a list of definitions followed by two lists of identifier mappings and a list of calibration mappings. The first list of identifier mappings is used to map from device measurement variables (eg. *xh*) to Planck input variables (eg. `x`). The second list of identifier mappings is used to map from Planck output variables (eg. `F`) to device output variables (eg. *force*). The list of calibration mappings is used to assign values to calibration parameters in the code (eg. to m).

3.4.1 Simplest Program

The simplest possible compiled program, shown in Figure 3.6, performs no computations but does serve to showcase the general structure of compiled programs.

```
{{}
 {}
 {}
 {}}
```

Figure 3.6: The simplest compiled program possible. It does nothing.

In more interesting programs (like those that follow), the first `{}` contains definitions, the second `{}` contains mappings from device measurement variables to Planck input variables, the third `{}` contains mappings from Planck output variables to device output variables. The last `{}` contains calibration mappings.

3.4.2 Spring

The following program simulates a spring on the Hapkit.

```
{{{k 100}
  {F {- {* k x}}}}
 {{xh x}
  {globalTime t}}
 {{F force}}
 {{rPulley 0.005}
  {rSector 0.077}
  {rHandle 0.074}
  {m 0.0002}
  {theta0 -0.1403}}}
```

Figure 3.7: A spring program for the Hapkit. The values of the calibration parameters are not necessarily the same for all Hapkits.

The program defines two identifiers `k` and `F` based on Hooke's Law shown in Equation 3.1. The value of `k` was chosen to given an aesthetically pleasing simulation. The device measurement variables *xh* and *globalTime* are mapped to `x` and `t` in the Planck simulation. The simulation variable `F` is mapped to the device output variable *force*. Finally, a number of calibration parameters are set based on the Hapkit used to run the simulation.

3.5 Running Planck Programs

The *Planck.rkt* file included in Appendix J defines a pair of functions:

```
(define (interpretProgram program)
(define (compileProgram program templateFilePath outFilePath)
```

The *interpretProgram* function consumes an interpreted Planck program and runs it. The *compileProgram* function consumes a compiled Planck program, the path to the template file and the file path at which to output the generated code.

Chapter 4

A Formal Introduction to Plank

The following chapter provides an overview of the Planck language as well as insight into the design goals, constraints and other languages that influenced Planck. Following this, a detailed examination of the language representation is given.

4.1 Overview

Planck is implemented in Racket using the PLAI Scheme [5] language extension. Interpreted Planck programs are interpreted within Racket. Compiled Planck programs use a Racket based code-generator to produce Arduino C code. Both of these types of programs share the mathematical core of the Plank language as well as the machinery for parsing and preprocessing programs to prepare them for interpretation and compilation.

All Planck program are treated as simulations. Time can be introduced explicitly through usage of a special identifier t in interpreted programs or through device bindings in compiled programs. It is also introduced implicitly through the *integrate* and *derivative* operators which perform numerical integration and differentiation respectively with respect to time.

4.2 Constraints

The main constraint affecting the the design of Planck arose from the high update frequency required by haptic systems in order to provide realistic tactile feedback. A prior project attempted to run the virtual environments on a host machine and send update parameters over USB to the Hapkit board. However the latency of communicating over USB resulted in a system that could not achieve the required update rate. This problem is addressed by compiling programs to Arduino C and running them directly on the Hapkit board. Using this, the update rate is limited only by the efficiency of the compiled code.

Another constraint lies in the fact that on the Arduino, the time interval between updates is not a constant. This affects the numerical schemes that can be used for integration and differentiation. Furthermore as the user is not restrained in terms of what they perform integration and differentiation on, the methods used must be able to handle a wide variety of cases. Both of these issues were discussed Section 2.2.

4.3 Influences

The design of Planck has been heavily influenced by FRAN [2], the functional reactive animation language of Conal Elliot. Much like FRAN focuses on separating the modeling of an animation from its presentation, Planck separates the modeling of a physical scenario from its simulation. It also draws inspiration from its treatment of time, which threads explicitly and implicitly through Planck programs.

Behaviors in FRAN, which are values that vary continuously with time, have a discrete analog within Planck which recomputes defined values at every iteration of the simulation. *Events* in FRAN, which refer to specific happenings in the world, have analogs within Planck in two main forms. The first form is conditional branching, which allows for programs to react to conditions within the simulation itself. The second form is user input, which allows for programs to react to conditions outside of the simulation.

The syntax of Planck is heavily influenced by the Lisp family of languages, mainly as a result of it having been implemented as a DSL within Racket. However future iterations of Planck need not necessarily utilize this syntax.

4.4 Language Representation

Planck's surface syntax is represented formally using an Extended Backus-Naur Form (EBNF). For interpretation and evaluation, Planck is parsed using Racket's *match* library and represented internally using the *define-type* capabilities of the PLAI Scheme language extension [5]. The internal representation has an almost one-to-one mapping with the EBNF of the language, with some exceptions. The full EBNF is located in Appendix A.

4.4.1 Programs - PROGRAM

```
<PROGRAM> ::= <program-I>
            | <program-C>
```

Planck can process two types of programs. `<program-I>` are interpreted programs whilst `<program-C>` are compiled programs. The EBNF for both of these program types follows

```
<program-I> ::= [(<DEFN> *)
                 (<IMAGE> *)
                 <num>
                 <num>]

<program-C> ::= [(<DEFN> *)
                 |  (<mapping> *)
                 |  (<mapping> *)
                 |  (<valMapping> *)]
```

Internally these are represented as a `PROGRAM` type with two variants.

```
(define-type PROGRAM
  (programI (loDefn (listof DEFN?))
            (loImage (listof IMAGE?))
            (worldClockRate number?)
            (tickSize number?))
  (programC (loDefn (listof DEFN?))
            (inputMappings (listof MAPPING?))
            (outputMappings (listof MAPPING?))
            (deviceValues (listof VAL-MAPPING?))))
```

Both interpreted and compiled programs have at their core a list of `DEFN`.

Interpreted programs have an additional list of `IMAGE` which provides information on how to render the simulation. The `worldClockRate` and `tickSize` determine the real-time between ticks and the size of the step size for numerical methods between ticks respectively. These need not be the same, resulting in simulations that run slower or faster than real time.

Compiled programs have two lists of mappings and a list of value mappings. The EBNF for `<mapping>` and `<val-mapping>` are

```
<mapping>     :: = (<id> <id>)

<val-mapping :: = (<id> <num>)
```

and their corresponding internal representation is

```
(define-type MAPPING
  (mapping (id1 symbol?) (id2 symbol?)))

(define-type VAL-MAPPING
  (val-mapping (id symbol?) (val num?)))
```

Each individual mapping is used to relate two identifiers. The first list of mappings is used to map device input identifiers to program input identifiers. The second list of mappings is used to map program output identifier to device output identifiers. The list of value mappings is used to bind identifiers corresponding to calibration parameters to numerical values.

4.4.2 Definitions - DEFN

Definitions are used to map between identifiers and expressions. Their EBNF form is

```
<DEFN> ::= (<id> <EXPR>)
```

and internally they are represented as

```
(define-type DEFN
  (defn (id symbol?) (expr EXPR?)))
```

4.4.3 Expressions - EXPR

Expressions are used to encode the mathematics of the scenario being simulated.

```
<EXPR> ::= (<num>)
         | (<bool>)
         | (<id>)
         | <unaryNumOp>
         | <binaryNumOp>
         | <binaryLogOp>
         | (integrate <EXPR>)
         | (derivative <EXPR>)
         | (if <EXPR> <EXPR> <EXPR>)
```

The base types for expressions are numbers (`<num>`) and booleans (`<bool>`). Identifiers (`<id>`) are used to refer to other definitions in the program. Unary numerical operators (`<unaryNumOp>`)

and binary numerical operators (`<binaryNumOp>`) provide access to the mathematical operators in the following EBNFs

```
<unaryNumOp> ::= (- <EXPR>)
               | (abs <EXPR>)
               | (sin <EXPR>)
               | (cos <EXPR>)
               | (tan <EXPR>
               | (sqr <EXPR>)
               | (sqrt <EXPR>)

<binaryNumOp> ::= (+ <EXPR> <EXPR>)
                | (- <EXPR> <EXPR>)
                | (* <EXPR> <EXPR>)
                | (/ <EXPR> <EXPR>)
```

Binary logical operators (`<binaryLogOperators>`) provide access to the mathematical operators in the following EBNF

```
<binaryLogOp> ::= (=   <EXPR> <EXPR>)
                | (<   <EXPR> <EXPR>)
                | (<=  <EXPR> <EXPR>)
                | (>   <EXPR> <EXPR>)
                | (>=  <EXPR> <EXPR>)
                | (and <EXPR> <EXPR>)
                | (or  <EXPR> <EXPR>)
```

which can be used with the conditional expression available.

Internally expressions have the following representation

```
(define-type EXPR
  [iden (id symbol?)]
  [num (n number?)]
  [bool (b boolean?)]
  [unaryNumOp (op procedure?) (expr EXPR?)]
  [binaryNumOp (op procedure?) (lexpr EXPR?) (rexpr EXPR?)]
  [binaryLogOp (op procedure?) (lexpr EXPR?) (rexpr EXPR?)]
  [integrate (expr EXPR?)]
  [derivative (expr EXPR?)]
  [_if (condExpr EXPR?) (trueExpr EXPR?) (falseExpr EXPR?)]
  [passForward])
```

Note the additional `passForward` expression which is used to handle and propagate external inputs and other special values without modification. The different types of mathematical operators are represented using a table that maps from the surface syntax for the operator to a Racket procedure that corresponds to the operator.

4.4.4 Rendering - IMAGE

The proof of concept implementation of Planck can only render circles in an interpreted context. Their EBNF follows.

```
<IMAGE> ::= (circle <EXPR-D> <string> <EXPR-D> <EXPR-D>)
```

Internally this is represented as

```
(define-type IMAGE
  (circ (rad EXPR-D?) (color string?) (x EXPR-D?) (y EXPR-D?)))
```

Circles are defined by a radius, a colour string, an x-position and a y-position. The values aside from the colour are populated using the `<EXPR-D>` type which has the following EBNF

```
<EXPR-D> ::= <id>
           | <num>
```

and internal representation

```
(define-type EXPR-D
  [idD (id symbol?)]
  [numD (n number?)])
```

`<EXPR-D>` can either be numbers or identifiers corresponding to the values of expressions bound by definitions.

Chapter 5

Preprocessing

While the surface syntax of Planck provides users with a large degree of flexibility in terms of how they write their programs, this flexibility becomes a hindrance when it comes to interpreting and compiling their programs. Thus before doing so a number of transformation are applied to take user programs and simplify them so that they utilize a subset of Planck that is easier to interpret and compile. These transformations all preserve the meaning of the programs. Various other preprocessing steps are also applied, such as adding identifiers for keyboard inputs to interpreted programs.

This chapter covers all of the preprocessing performed on Planck programs and includes examples of preprocessing for both interpreted and compiled programs. Note that the preprocessing steps are applied to the internal representation of the Planck programs and not the surface representation.

5.1 Desugaring

Conditional branching, integration and differentiation operators become difficult to reason about when they are nested within other expressions. Furthermore, compiling the components of these expressions can result in various special cases. To avoid these cases and simplify interpretation and compilation, the various subexpression of these expressions are converted into identifiers.

5.1.1 Conditional Desugaring

Nested conditional expressions are moved to new definitions, generating a unique new identifier for each expression moved. The following transformation is then applied, again creating new identifiers as necessary.

$$\texttt{(if <expr> <expr> <expr>)} \longrightarrow \texttt{(if <id> <id> <id>)} \tag{5.1}$$

Thus the following code

```
{x {abs {if {< 1 2} 1 2}}}
```

would first be transformed into the equivalent form

```
{x {abs y}}
{y {if {< 1 2} 1 2}}
```

and finally to

```
{x {abs y}}
{y {if a b c}}
{a {< 1 2}}
{b 1}
{c 1}
```

5.1.2 Integration and Derivative Desugaring

Nested integration and derivative expressions are moved to new definitions, generating a unique new identifier for each expression moved. The following transformations are then applied, again creating new identifiers as necessary.

$$\texttt{(integrate <expr>)} \longrightarrow \texttt{(integrate <id>)} \tag{5.2}$$

$$\texttt{(derivative <expr>)} \longrightarrow \texttt{(derivative <id>)} \tag{5.3}$$

Thus the following code

```
{a {- {integrate 1}}}
{x {+ {derivative 2} 3}}
```

would first be transformed to the equivalent form

```
{a {- b}
{b {integrate 1}}
{x {+ y 3}}
{y {derivative 2}}
```

and finally to

```
{a {- b}
{b {integrate c}}
{c 1}
{x {+ y 3}}
{y {derivative z}}
{z 2}
```

After this transformation all derivative expressions are of the form

$$y = \frac{dx}{dt} \tag{5.4}$$

and all integration expressions are in the form

$$y = \int_0^t x\,dt \tag{5.5}$$

where x and y are arbitrary identifiers. This form simplifies the application of the Forward Difference method and Forward Euler method for computing numerical derivatives and integrals respectively as will be demonstrated in Chapters 6 and 7

5.2 Identifier Injection

Both interpreted and compiled programs have special expressions and identifiers that need to be injected into them in order to work properly. Identifiers are injected by adding them to the list of definitions generated from parsing the program.

Interpreted programs have multiple identifiers injected into them. The identifier `t` is bound to the current time. It is injected using the expression `(integrate 1)` which corresponds mathematically to the value of the current time as per Equation 5.6.

$$\int_0^t 1\,dt = t|_0^t = t \tag{5.6}$$

The identifier `delta_t`, which corresponds to the step-size between time points, is bound to the value given by `tickSize` in the program and is used by the Planck interpreter when applying numerical differentiation and integration. The identifiers `ku`, `kd`, `kr` and `kl` are bound to `passForward` expressions. These identifiers corresponds to the four arrow keys on keyboards.

Compiled programs have the program input identifiers injected into them, bound to the `passForward` expression. For example the Hapkit spring program in Figure 3.7 has the identifiers `x` and `t` injected into it, which allows the program to utilize the values of *xh* and *globalTime* from the template respectively.

5.3 Preprocessing an Interpreted Program

The program used for this example is the spring with keyboard input from Figure 3.5. After parsing this program, the internal representation of its list of definitions is that in Figure 5.1.

```
'[(defn x  (binaryNumOp + (integrate v) (num 100)))
  (defn v  (integrate (iden F)))
  (defn F  (unaryNumOp - (binaryNumOp * (num 200) (iden x))))
  (defn R  (binaryNumOp + (binaryNumOp * (num 40) (iden ku))
                          (num 10)))]
```

Figure 5.1: The list of definitions from the interpreted spring program after parsing.

The preprocessing pass over the program adds in all of the special identifiers needed for an interpreted program and also handles the integrate expression within the definition for `F` which does not have the correct form. The resulting list of definitions is found in Figure 5.2

```
'[(defn delta_t (passForward))
  (defn ku  (passForward))
  (defn kd  (passForward))
  (defn kl  (passForward))
  (defn kr  (passForward))
  (defn t   (integrate _t1))
  (defn _t1 (num 1))
  (defn x   (binaryNumOp + (iden _x2) (num 100)))
  (defn _x2 (integrate (iden v)))
  (defn v   (integrate (iden F)))
  (defn F   (unaryNumOp - (binaryNumOp * (num 200) (iden x))))
  (defn R   (binaryNumOp + (binaryNumOp * (num 40) (iden ku))
                           (num 10)))]
```

Figure 5.2: The list of definitions from the interpreted spring program after preprocessing.

5.4 Preprocessing a Compiled Program

The program used for this example is the spring program from Figure 3.7. The list of definitions after parsing is shown in Figure 5.3.

```
'[(defn k  (num 100))
  (defn F (unaryNumOp - (binaryNumOp * (iden k) (iden x))))]
```

Figure 5.3: The list of definitions from the compiled spring program after parsing.

Preprocessing of the program adds in the program input identifiers, resulting in the list of definitions in Figure 5.4.

```
'[(defn k  (num 100))
  (defn F (unaryNumOp - (binaryNumOp * (iden k) (iden x))))
  (defn x (passForward))
  (defn t (passForward))]
```

Figure 5.4: The list of definitions from the compiled spring program after preprocessing.

Chapter 6

Interpretation

6.1 Interpretation Strategy

The interpretation of programs is supported by two main structures, an expression store and a world store.

The expression store is initialized once from the list of definitions in a program. It is an associative mapping from identifiers to the internal representations of their corresponding expressions. This same store is used throughout the entire interpretation. In the current Planck interpreter, the expression store is implemented using a hash table.

The world store is an associative mapping from identifiers to their values in the current time of the simulation. In each iterative step of the simulation, a new world is created and populated by reinterpreting all of the expressions in the expression store. The exception to this are the keyboard input identifiers `ku`, `kd`, `kl` and `kr` whose value in the world do not come from the simulation but from the physical state of the keys they correspond to.

The flow of interpretation is captured in Figure 6.1. The sections that follow in this chapter will give details on each of the simulation phases.

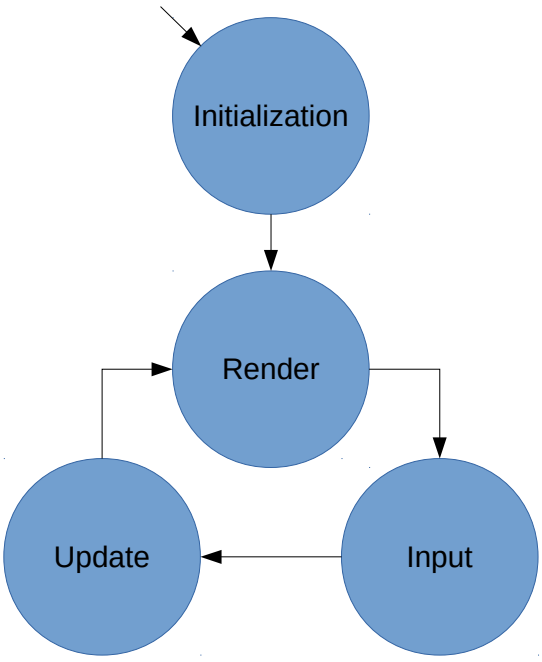

Figure 6.1: The control flow of program interpretation. Each circle corresponds to a different phase.

The interpretation of the program is aided by the *universe* [3] library. The *big-bang* form available from this library provides a simple construct for running simulations. Using the form requires defining a number of event handlers and setting a tick rate for the simulation. The *on-tick* handler is triggered at every tick, and is associated with the creation and population of a new world. The *on-draw* handler is used to render the world at every tick. Finally the *on-key* and *on-release* handlers are used to detect key events and modify the world directly to update the values of `ku`, `kd`, `kl` and `kr`.

The *on-tick* handler corresponds to the update phase, the *on-draw* handler to the render phase and the *on-key* and *on-release* handlers to the input phase.

6.1.1 Initialization Phase

The initialization phase begins by populating the expression store from the preprocessed programs list of definitions. Following this, every expression in the expression store is interpreted for the first time. A special property of this interpretation is that all integrations are evaluated to 0 based on the result of Equation 6.1.

$$\int_0^0 f(t)\,dt = 0 \tag{6.1}$$

In addition all derivatives are evaluated to 1 (an arbitrary choice, but one which quickly corrects itself).

During the interpretation of an expression it is possible that it depends on a value that is not currently available in the world. In such cases interpretation of the current expression is halted and interpretation of the expression it depends on begins. During this process, care is taken to keep track of which expressions are in the process of being interpreted.

If a dependency cycle is detected the interpreter halts and throws a Racket error. Note that dependency cycles are cut by integration in this phase as all integration operations evaluate to 0 and have no need to evaluate the integrand. For example the program in Figure 6.2 contains a cycle because `x` requires `y` and vice-versa. However the program in Figure 6.3 does not have a cycle as `y` depends on the integral of `x`.

```
{{{x y}
  {y x}}
 {}
 1
 1}
```

Figure 6.2: An interpreted program with a cycle.

```
{{{x y}
  {y {integrate x}}}
 {}
 1
 1}
```

Figure 6.3: An interpreted program without a cycle.

The result of the initialization phase is a complete world store.

6.1.2 Render Phase

During the render phase, the render expressions defined in the program are rendered using the state of the current world. Rendering is achieved using the *images* [7] library to draw shapes onto a canvas. Any identifier references used within the render expressions are looked up in the current world.

Both the x and y-axes of the canvas support a range of values from -500 to 500. The positions of render expressions are given in Cartesian coordinates relative to an origin at the center of this canvas. However in practice the Racket canvas uses an origin at the top-left corner. To handle this, positions in render expressions are passed through an additional transformation to convert coordinates as needed.

6.1.3 Input Phase

In the input phase, the states of the relevant keyboard inputs are determined. The values associated with the keyboard identifiers in the current world are then mutated to match their physical state.

6.1.4 Update Phase

The update phase begins with the creation of an entirely new world store. Much like in the initialization phase, every expression in the expression store is interpreted and its result stored in the world store. The main difference is that in the update phase integration is now performed, using results from the previous world to populate the current world. At any point in time at most two worlds exist, and once the update is complete the previous world is destroyed.

Integration

In the update phase, Forward Euler integration of the following definition

```
(y (integrate x))
```

corresponds to

$$y_{\text{Current World}} = y_{\text{Previous World}} + \Delta t \times x_{\text{Previous World}}$$

where Δt corresponds to the value of `delta_t`. The value of `y` in the current world depends solely on values in the previous world.

Differentiation

In the update phase, Forward Difference differentiation of the following definition

```
(y (differentiate x))
```

corresponds to

$$y_{\text{Current World}} = \frac{x_{\text{Current World}} - x_{\text{Previous World}}}{\Delta t}$$

where Δt corresponds to the value of `delta_t`. Unlike with integration, the value of `y` in the current world depends on values in both the current and previous world. Furthermore the previous value of `y` is not required for the computation.

6.1.5 Example

In order to provide insight into how interpretation works, the spring with keyboard input program in Figure 3.5 will be interpreted by hand for two cycles. The interpreter makes use of the preprocessed form of the program in Figure 5.2.

Interpretation begins in the initialization stage and starts by creating an expression store as shown in Figure 6.4. The expression store has a direct one-to-one mapping with the preprocessed list of definitions.

```
'[(delta_t : (passForward))
  (ku  : (passForward))
  (kd  : (passForward))
  (kl  : (passForward))
  (kr  : (passForward))
  (t   : (integrate _t1))
  (_t1 : (num 1))
  (x   : (binaryNumOp + (iden _x2) (num 100)))
  (_x2 : (integrate (iden v)))
  (v   : (integrate (iden F)))
  (F   : (unaryNumOp - (binaryNumOp * (num 200) (iden x)))
  (R   : (binaryNumOp + (binaryNumOp * (num 40) (iden ku))
                        (num 10)))]
```

Figure 6.4: The expression store associated with the program.

Continuing in the initialization phase, the initial interpretation of all of the expressions in the expression store yields the world in Figure 6.5. Note that this assumes that all of the arrow keys are depressed. Notice that all of the identifiers bound to `integrate` expressions (ie. `t`, `_x2` and `v`) evaluate to 0 in this stage as expected. With the initial world in place, interpretation moves

```
[(delta_t : 0.001) (ku : 0) (kd : 0) (kl : 0) (kr : 0)
 (t   : 0)
 (_t1 : 1)
 (x   : 100)
 (_x2 : 0)
 (v   : 0)
 (F   : -20000)
 (R   : 10)]
```

Figure 6.5: The world at $t = 0$.

to the first pass of the render phase which generate the frame in Figure 6.8 based on the world state.

From here interpretation continues to the input phase, where it is assumed that no arrows keys are depressed, and then the first update phase. During the update phase the world in Figure 6.6 is generated. The values of integrals in this world are computed as follows

```
[(delta_t : 0.001) (ku : 1) (kd : 0) (kl : 0) (kr : 0)
 (t    : 0.001)
 (_t1  : 1)
 (x    : 100)
 (_x2  : 0)
 (v    : -20)
 (F    : -20000)
 (R    : 10)]
```

Figure 6.6: The world at $t = 0.001$.

$$
\begin{aligned}
t &\leftarrow t + \Delta t \times \_t1 = 0 + 0.001 \times 1 = 0.001 \\
v &\leftarrow v + \Delta t \times F = 0 + 0.001 \times -20000 = -20 \\
\_x2 &\leftarrow \_x2 + \Delta t \times v = 0 + 0.001 \times 0 = 0
\end{aligned}
$$

When this world is rendered, it produces the frame in Figure 6.9. Note how the blue circle has moved down in response to the decrease in velocity.

After this the interpreter enters the input phase once again, but this time let us assume that the up arrow key is depressed. After the completion of the following update phase, the world in Figure 6.7 is generated. Note how the value of `R` changes in response to the keyboard input. The

```
[(delta_t : 0.001) (ku : 0) (kd : 0) (kl : 0) (kr : 0)
 (t    : 0.002)
 (_t1  : 1)
 (x    : 99.98)
 (_x2  : -0.02)
 (v    : -40)
 (F    : -19996)
 (R    : 50)]
```

Figure 6.7: The world at $t = 0.002$.

effect of this can also be seen in the output of the render phase in Figure 6.10 in which the red circle has expanded.

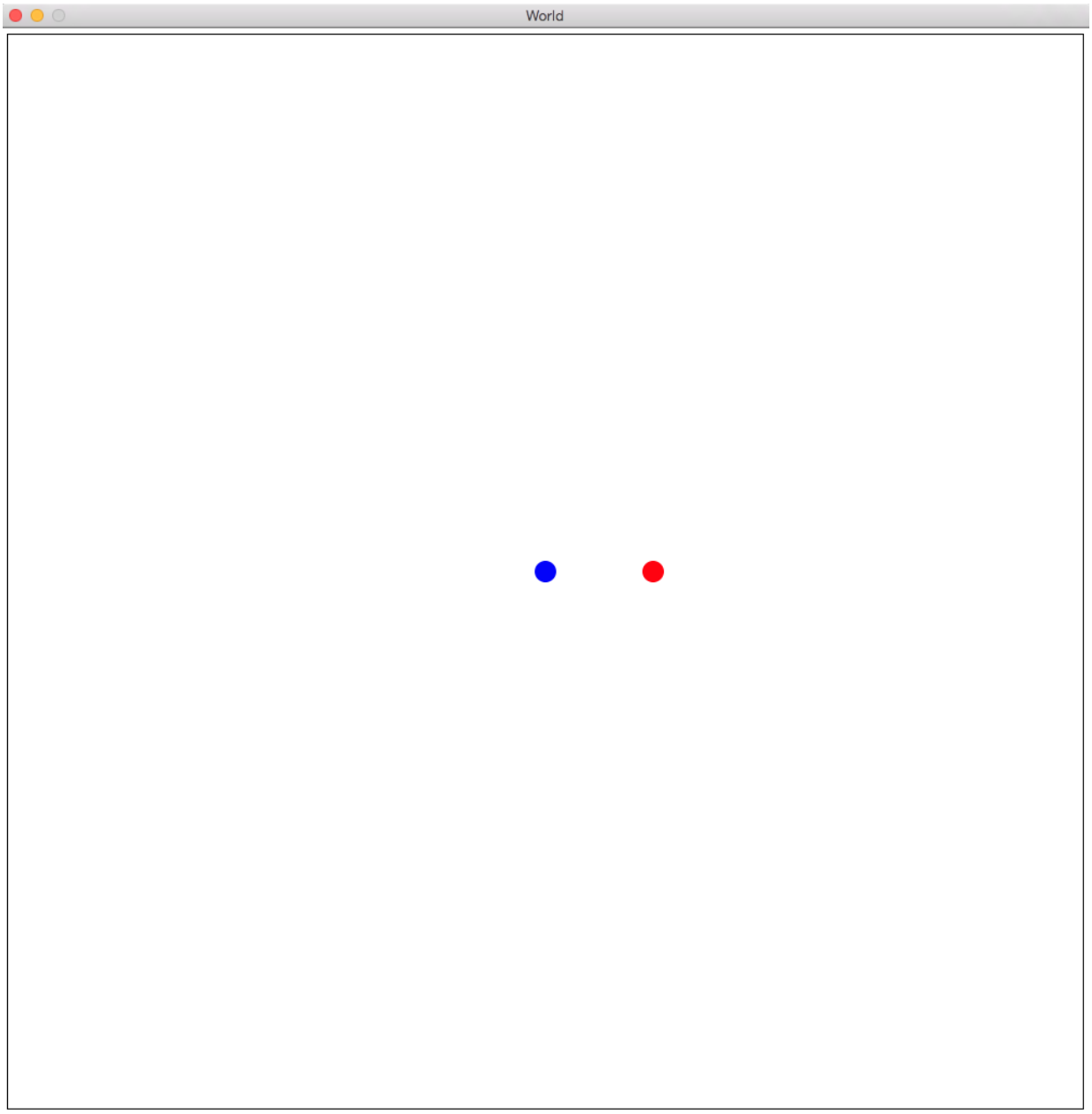

Figure 6.8: The world rendered at $t = 0.000$

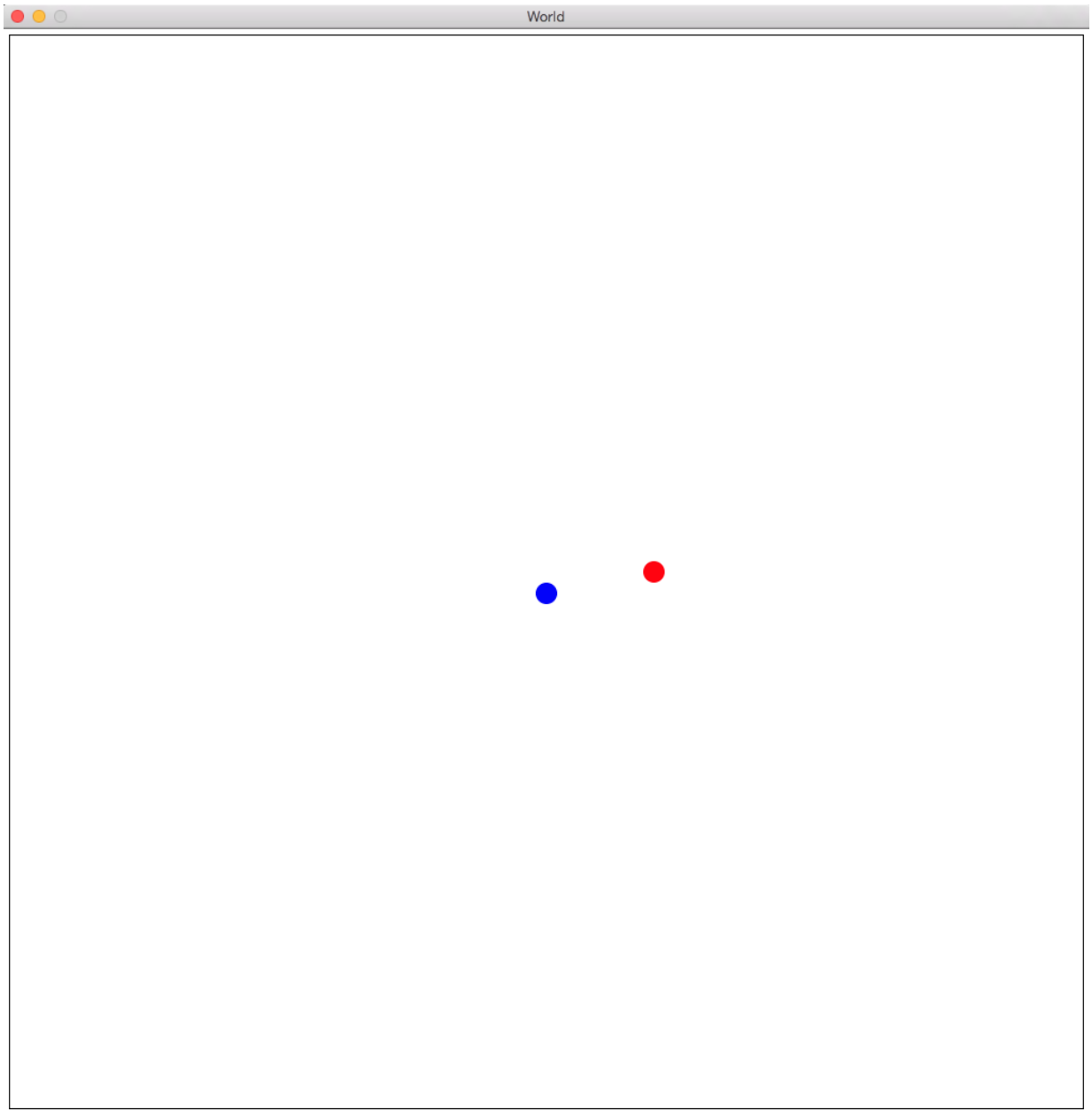

Figure 6.9: The world rendered at $t = 0.001$

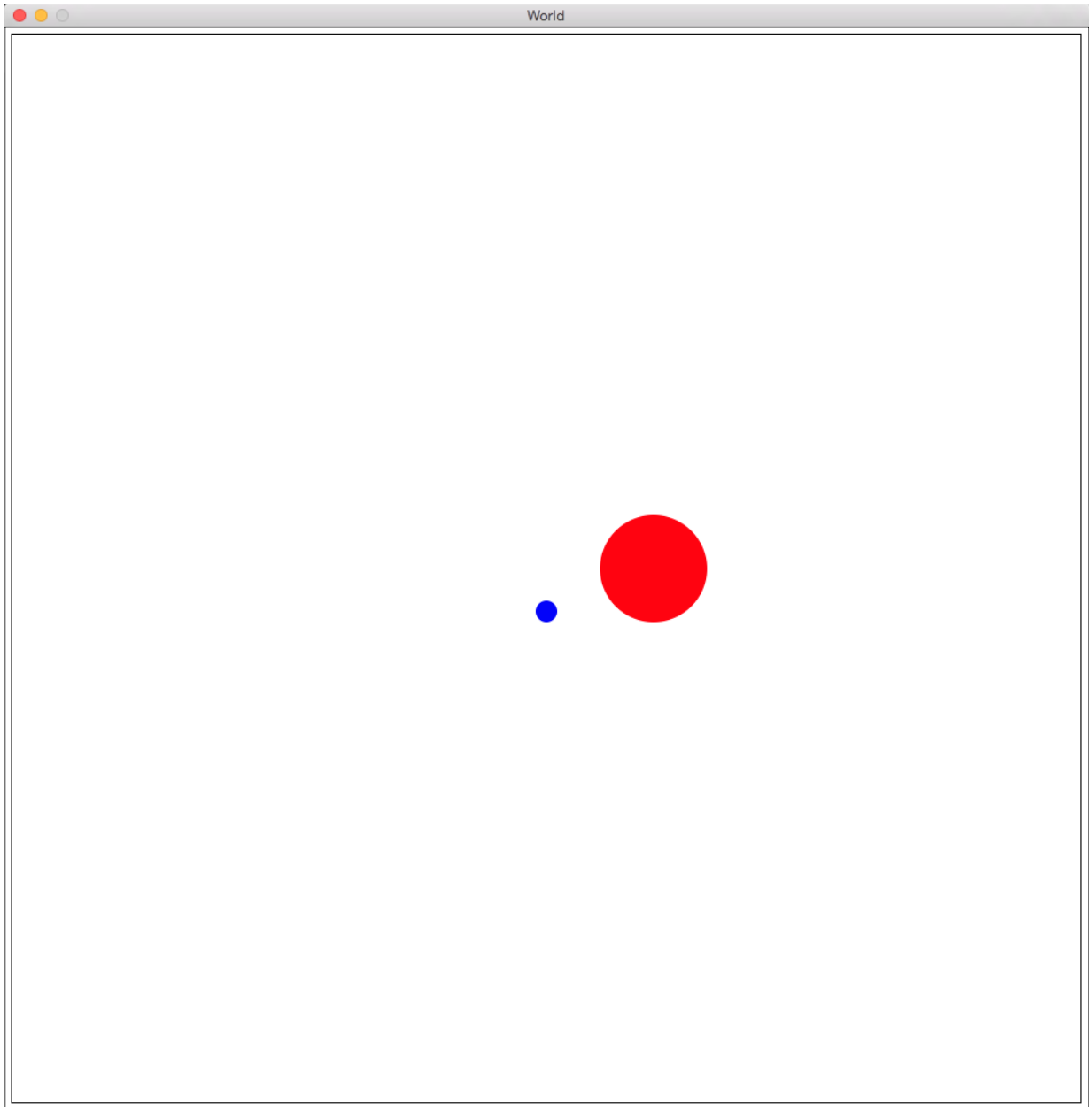

Figure 6.10: The world rendered at $t = 0.002$

Chapter 7

Compilation

This chapter covers the the code generation strategy used to take a Planck program and convert it to Arduino C. It walks through the structure of the general template file and shows how the different sections of the template are filled out.

7.1 Templates

Thus far it has been assumed that the Hapkit was the only target for compilation. However Arduino boards could be used to create a wide variety of haptic devices. In order to have Planck target all of these devices it is beneficial to think in terms of a general code template rather than a Hapkit specific template. This general template has a common schema that only differs when defining device specific I/O functions. In order to use a device as a target for Planck compilation, all that would be required is that it have a valid template associated with it.

In terms of their implementation, templates are written as large format strings with multiple segments for inserting code. The inserted code is generated by passing over the internal representation of the program being compiled multiple times.

The simulation model used within the templates is mostly a direct mapping of the simulation strategy used by the interpreter. All templates begin with a structure like that in Figure 7.1. The details of how this is filled in follow.

7.1.1 Imports

All templates begin by importing any required C libraries.

7.1.2 Definitions

The world store in the interpreted program is mapped to an array in the compiled version. To allow for constant time lookup, each identifier is mapped to a unique integer starting from 0. These values are then encoding using C macros. For example the following program

```
{x {abs 1}}
{y {+ x 2}}
{z {- y}}
```

generates the following C code for the *DEFINES* segment of the template.

```
#define x 0
#define y 1
#define z 2
```

```
// IMPORTS
#include <arduino.h>

// DEFINES
~a

// TIME
double globalTime;
double prevTime;
double deltaT;

// WORLD
double worldStore1[NumOfStates] = {0};
double worldStore2[NumOfStates] = {0};
bool notUpdated[NumOfStates] = {0};
double *prevWorld;
double *currWorld;

// INTEGRATION AND DIFFERENTIATION
double integrate(int variableKey, int integrandKey) {
    double oldVariableValue = prevWorld[variableKey];
    double oldIntegrandValue = prevWorld[integrandKey];
    return oldVariableValue + deltaT * oldIntegrandValue;
}

double derivative(int derivativeKey) {
    return  (currWorld[derivativeKey] - prevWorld[derivativeKey])
       / deltaT;
}

// PROTOTYPES
~a
~a

// STATE INITIALIZATION FUNCTIONS
~a

// STATE FUNCTIONS
~a
```

Figure 7.1: The start of a template file. `~a` corresponds to segments of the template that are populated by the code generator.

7.1.3 Time

The *TIME* segment of the template is used to keep track of time related values. The global time, previous iteration time and the change in time between the previous and current iteration are all stored in these variables. This last quantity is especially important as it is used when performing numerical differentiation and integration.

7.1.4 World

The *WORLD* segment of the template is used to store the state of the world in the simulation. Rather than dynamically allocating a new world during the update phase, templates define two arrays to use for world storage. Two additional pointers to these array are also defined, one which is used to point to the current world and the other to the previous world. At the beginning of every update phase these are swapped. In order to keep track of which values in the current world need to be updated, an additional boolean array is used. This array is reset at the beginning of every update phase. All of these arrays can be indexed using identifier names (which are now equivalent to integers by the macros generated previously).

7.1.5 Integration and Differentiation

The *INTEGRATION AND DIFFERENTIATION* segment holds the functions `integrate` and `derivative` which perform Forward Euler integration and Forward Difference differentiation using the world state. These implementations are direct analogs to the implementations in the interpreter.

7.1.6 Expressions

Rather than storing expressions in an expression store, each individual definition in the program is turned into a function. For example the following definition

```
{x {+ 1 y}}
```

becomes

```
double State_x() {
    if (notUpdated[x]) {
        currWorld[x] = (1 + State_y());
        notUpdated[x]=0;
    }
    return currWorld[x];
}
```

Looking up an identifier in the current world is replaced with calling the function associated with the identifier. These functions also handle the logic of checking whether the value needed has been computed. Definitions involving integration, derivatives or conditional branches are handled as special cases.

Integration

In the compiled program, the definition

```
{y {integrate x}}
```

corresponds to the function

```
double State_y() {
    if (notUpdated[y]) {
        currWorld[y] = integrate(y, x);
        notUpdated[y]=0;
    }
    return currWorld[y];
}
```

Differentiation

In the compiled program, the definition

```
{y {derivative x}}
```

corresponds to the function

```
double State_y() {
    if (notUpdated[y]) {
        currWorld[y] = derivative(x));
        notUpdated[y] = 0;
    }
    return currWorld[y];
}
```

Conditionals

In the compiled program, the definition

```
{y {if a b c}}
```

corresponds to the function

```
double State_y() {
    if (notUpdated[y]) {
        if (State_a) {
            currWorld[y] = State_b();
        } else {
            currWorld[y] = State_c();
        }
        notUpdated[y] = 0;
    }
    return currWorld[y];
}
```

Pass Forward

In the compiled program, the internal definition

```
(x (passForward))
```

corresponds to the function

```
double State_x() {
    return currWorld[x];
}
```

In compiled programs, the `passForward` expression is used solely with device inputs. These inputs are inserted into the current world at the beginning of each iteration, thus identifiers bound to `passForward` can be retrieved directly from the current world.

7.1.7 Protoypes

The *PROTOTYPES* segment is used to define function prototypes for of all the identifier functions. This is done to prevent compiler errors when compiling the generated Arduino C code, as identifier functions earlier in the generated code may depend on identifier functions that are not defined until later.

7.1.8 State Initialization and State Functions

For each definition, the code generator produces two functions. The first set of these functions are the initialization functions which all have the prototype

```
double InitState_<id>();
```

followed by the state functions

```
double State_<id>();
```

The bodies of both of these types of functions are very similar. The only difference arises when handling integration and differentiation. All of the *InitState* functions simply return 0 for `integrate` expressions and 1 for `derivative` expressions. The *State* functions on the other hand call the `integrate` and `derivative` helper functions. This corresponds to the difference between interpretation in the initialization phase and the update phase.

The functions generated are inserted into the *STATE INITIALIZATION FUNCTIONS* and *STATE FUNCTIONS* segments of the template.

7.1.9 Device Specific Code

The device specific code section of the template (shown in Figure 7.2) holds the code that reads raw device inputs and convert them into reasonable physical values for the Planck program to consume. It also has code for taking values from the Planck program and translating them into values for device outputs.

This section must also contain declarations for all device output identifiers and calibration value identifiers.

```
// ----- Device Specific Code Starts -----
// Declare parameters for output mappings.
  double deviceOutputValue; // Example
// Helper functions for handling device I/O
// Special device parameters;
  double calibrationParam; // Example
// ----- Device Specific Code Ends   -----
```

Figure 7.2: The device specific portion of the template file. The defined identifiers are just general examples.

7.1.10 setup

All Arduino programs have a special function *setup* which is called once at the beginning of program execution. This corresponds roughly to the initialization phase of the interpreter. Figure 7.3 shows the setup section of the template. The first section to fill in is used to set device calibration parameters. For example if the program's list of value mappings contained

```
{calibrationParam 42}
```

then the following code would be generated

```
calibrationParam = 42;
```

After this section there is a block of code used to initialize all of the time variables as well as the world arrays.

This initialization is followed by a block of device specific code that reads in sensor data. By the end of this block of code, the template must have assigned values to all of the device input identifiers. For example in the context of the spring program in Figure 3.7, by the end of this block the device input identifer *xh* must have been assigned a value.

The next section is one to fill in. Here all of the program input identifiers are assigned the device input identifiers. For the spring program this segment would contain the following code.

```
currWorld[x] = xh;
currWorld[t] = globalTime;
```

Then all of the *InitState* functions are called to force the simulation to populate the current world. For the spring program this looks like

```
InitState_F();
InitState_k();
InitState_x();
InitState_t();
```

Finally all of the device output identifier are assigned values from the world. For the spring program this looks like.

```
force = InitState_F();
```

In the final block of device specific code, the device output identifiers are used to control device outputs.

```
void setup() {
    // Initialize Device Values
    ~a

    globalTime = 0;
    prevTime = micros();
    prevWorld = worldStore1;
    currWorld = worldStore2;
    memset(notUpdated, 1, NumOfStates * sizeof(bool));

    // ----- Device Specific Code Starts -----
    // Read sensor inputs into device input identifiers.
    // ----- Device Specific Code Ends   -----

    // Assign device input identifiers to program input
       identifiers.
    // Call all InitState functions.
    // Assign program output identifiers to device output
       identifiers.
    ~a

    // ----- Device Specific Code Starts -----
    // Handle device outputs.
    // ----- Device Specific Code Ends   -----
```

Figure 7.3: The setup function. It is the first function called by all Arduino programs.

```
void loop() {
    double tNow = micros();
    deltaT = tNow - prevTime;
    prevTime = tNow;
    globalTime += deltaT;
    double* temp = prevWorld;
    prevWorld = currWorld;
    currWorld = temp;
    memset(notUpdated, 1, NumOfStates * sizeof(bool));

    // ----- Device Specific Code Starts -----
    // Read sensor inputs into device sensor identifiers.
    // ----- Device Specific Code Ends   -----

    // Assign device input identifiers to program input
       identifiers.
    // Call all State functions.
    // Assign program output identifiers to device output
       identifiers.
    ~a
    // ----- Device Specific Code Starts -----
    // Handle device outputs.
    // ----- Device Specific Code Ends   -----
}
```

Figure 7.4: The loop function. It is called continuously over execution.

7.1.11 loop

All Arduino programs have a special function *loop* which is called continuously during execution. This corresponds roughly to the simulation cycle of the interpreter. Figure 7.4 shows the loop section of the template.

At the beginning of the loop the current time is computed. Based on the previous time measured, this is used to determine the size of the time step and update *delta_t*. The beginning of the loop also swaps the world arrays and resets the boolean update array.

After this section the rest of the *loop* function actually mirrors the *setup* function. The only difference is that all references to *InitState* functions are replaced by *State* functions.

Chapter 8

Conclusion

The proof-of-concept implementation of the Planck language demonstrates that it is possible to create a language abstraction that allows users to write succinct programs modeling mathematical and physical scenarios whilst hiding away the details of how and where they are executed. As is Planck can generate basic visualizations and generate code for the Hapkit. However there is much room to improve Planck.

Adding in more mathematical operators like vectors, cross products and dot products would allow for succinct representations of more complex physical phenomena which are often described using these operators.

The current syntax of Planck is very Lispish, however other front-ends could be created to allow for other syntaxes or even graphical programming capabilities.

Expanding on the rendering aspect of the language would allow users to create richer visualizations for their simulations. In fact even compiled programs could have access to the rendering sub-language. Arduino boards have the ability to communicate to a host machine via USB. Taking advantage of this, compiled programs could send program values to the host machine at each iteration. These values could then be used with the rendering language (or anything that can read serial input) to provide visualizations on the host machine. With this, a Planck virtual environments could be explored both tactually and visually.

Improving the performance of the Planck simulation loop would allow user to encode larger simulations and also improve their accuracy. The accuracy of most numerical methods depends on the step-size used by the method. In Planck simulations, this step-size is proportional to the duration a single loop iteration. Thus improving the loop performance would have the benefit of increasing the simulation accuracy. Currently much time is wasted during evaluation (in both interpreted and compiled Planck) checking for whether values have or have not been computed yet. It is possible to do away with these checks by ordering expression evaluation intelligently. Such an order can be determined by generating a dependency graph of all of the expressions and then determining a topological ordering.

Accuracy can also be improved directly by improving the quality of the numerical methods used. The current methods, while versatile, are not the most accurate. Thus finding and implementing more accurate methods would be beneficial. However there may not be a single solution for all possible cases, in which case performing additional program analysis to match different classes of integrals and derivatives to different methods would be very beneficial.

One improvement that should be made is the addition of identifier mangling for compiled programs. With the current implementation it is possible that an identifier in a definition will clash with an identifier in a template. This can be remedied quite easily be restricting usage of underscores at the front of user-define identifiers for the surface language, and then prepending underscores to all program identifiers before compilation. Templates would then have to be written so as to *not* use identifiers with underscores at the start.

Bibliography

[1] Arduino. Arduino, 2014. [Online; accessed 29-March-2015].

[2] Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273, August 1997.

[3] Matthias Felleisen. Worlds and the universe, 2014. [Online; accessed 31-March-2015].

[4] National Science Foundation. Cyberlearning and future learning technologies, 2014. [Online; accessed 18-March-2015].

[5] Racket Language. Plai scheme, 2014. [Online; accessed 28-March-2015].

[6] University of Colorado Boulder. Phet interactive simulations, 2015. [Online; accessed 13-April-2015].

[7] Racket. Images, 2014. [Online; accessed 31-March-2015].

[8] Stanford. About hapkit, 2015. [Online; accessed 14-April-2015].

[9] Stanford. Hapkit image, 2015. [Online; accessed 29-March-2015].

[10] Stanford. Introduction to haptics, 2015. [Online; accessed 14-April-2015].

Appendix A

Extended BNF

The full EBNF of Planck follows.

```
<PROGRAM> ::= <program-I>
            | <program-C>

<program-I> ::= [(<DEFN> *)
                 (<IMAGE> *)
                 <num>
                 <num>]

<program-C> ::= [(<DEFN> *)
              |  (<mapping> *)
              |  (<mapping> *)
              |  (<valMapping> *)]

<mapping>    :: = (<id> <id>)

<val-mapping :: = (<id> <num>)

<DEFN> ::= (<id> <EXPR>)

<EXPR> ::= (<num>)
         | (<bool>)
         | (<id>)
         | <unaryNumOp>
         | <binaryNumOp>
         | <binaryLogOp>
         | (integrate <EXPR>)
         | (derivative <EXPR>)
         | (if <EXPR> <EXPR> <EXPR>)

<unaryNumOp> ::= (- <EXPR>)
               | (abs <EXPR>)
               | (sin <EXPR>)
               | (cos <EXPR>)
               | (tan <EXPR>
               | (sqr <EXPR>)
               | (sqrt <EXPR>)
```

```
<binaryNumOp> ::= (+ <EXPR> <EXPR>)
                | (- <EXPR> <EXPR>)
                | (* <EXPR> <EXPR>)
                | (/ <EXPR> <EXPR>)

<binaryLogOp> ::= (=   <EXPR> <EXPR>)
                | (<   <EXPR> <EXPR>)
                | (<=  <EXPR> <EXPR>)
                | (>   <EXPR> <EXPR>)
                | (>=  <EXPR> <EXPR>)
                | (and <EXPR> <EXPR>)
                | (or  <EXPR> <EXPR>)

<IMAGE> ::= (circle <EXPR-D> <string> <EXPR-D> <EXPR-D>)

<EXPR-D> ::= <id>
           | <num>
```

Appendix B

Language.rkt

The full internal Racket representation of Planck follows.

```
#lang racket

(provide (all-defined-out))

(require plai/datatype)

(define unaryNumOpTable
  (hash '-    -
        'abs  abs
        'sin  sin
        'cos  cos
        'tan  tan
        'sqr  sqr
        'sqrt sqrt))

(define binaryNumOpTable
  (hash '+ +
        '- -
        '* *
        '/ /))

(define (_and expr1 expr2)
  (and expr1 expr2))

(define (_or expr1 expr2)
  (or expr1 expr2))

(define binaryLogOpTable
  (hash '=   =
        '<   <
        '<=  <=
        '>   >
        '>=  >=
        'and _and
        'or  _or))

(define-type EXPR
```

```
  [iden (id symbol?)]
  [num (n number?)]
  [bool (b boolean?)]
  [unaryNumOp (op procedure?) (expr EXPR?)]
  [binaryNumOp (op procedure?) (lexpr EXPR?) (rexpr EXPR?)]
  [binaryLogOp (op procedure?) (lexpr EXPR?) (rexpr EXPR?)]
  [integrate (expr EXPR?)]
  [derivative (expr EXPR?)]
  [_if (condExpr EXPR?) (trueExpr EXPR?) (falseExpr EXPR?)]
  [passForward])

(define-type DEFN
  (defn (id symbol?) (expr EXPR?)))

(define-type MAPPING
  (mapping (id1 symbol?) (id2 symbol?)))

(define-type VAL-MAPPING
  (val-mapping (id symbol?) (val num?)))

(define-type PROGRAM
  (programI (loDefn (listof DEFN?))
            (loImage (listof IMAGE?))
            (worldClockRate number?)
            (tickSize number?))
  (programC (loDefn (listof DEFN?))
            (inputMappings (listof MAPPING?))
            (outputMappings (listof MAPPING?))
            (deviceValues (listof VAL-MAPPING?))))

;; Rendering
(define-type EXPR-D
  [idD (id symbol?)]
  [numD (n number?)])

(define-type IMAGE
  (circ (rad EXPR-D?) (color string?) (x EXPR-D?) (y EXPR-D?)))

;; Reserved Symbols
(define reserved-symbols '(integrate derivative if TRUE FALSE
   delta_t))

;; valid-identifier? : any -> boolean
;; Determines whether the parameter is a valid identifier name.
(define (valid-identifier? s)
  (and (symbol? s)
```

```
    (not (member s reserved-symbols))
    (not (hash-has-key? unaryNumOpTable s))
    (not (hash-has-key? binaryNumOpTable s))
    (not (hash-has-key? binaryLogOpTable s))))
```

Appendix C

Parser.rkt

The following code is used to parse Planck programs.

```
#lang racket

(provide (all-defined-out))

(require "Language.rkt")

;; parseExpr : any -> EXPR
;; Consumes an s-expression and generates an EXPR.
(define (parseExpr sexp)
  (match sexp
    [(? valid-identifier?) (iden sexp)]
    [(? number?) (num sexp)]
    [(list (? (lambda (op) (hash-has-key? binaryNumOpTable op)) op
       )
           lexp rexp)
     (binaryNumOp (hash-ref binaryNumOpTable op)
             (parseExpr lexp)
             (parseExpr rexp))]
    [(list (? (lambda (op) (hash-has-key? binaryLogOpTable op)) op
       )
           lexp rexp)
     (binaryLogOp (hash-ref binaryLogOpTable op)
             (parseExpr lexp)
             (parseExpr rexp))]
    [(list (? (lambda (op) (hash-has-key? unaryNumOpTable op)) op)
           expr)
     (unaryNumOp (hash-ref unaryNumOpTable op)
            (parseExpr expr))]
    [`(integrate ,expr) (integrate (parseExpr expr))]
    [`(derivative ,expr) (derivative (parseExpr expr))]
    [`(if ,condExpr ,trueExpr ,falseExpr)
      (_if (parseExpr condExpr)
           (parseExpr trueExpr)
           (parseExpr falseExpr))]
    [else (error 'parseExpr "Invalid␣expr:␣~s" sexp)]))

;; parseDefn : any -> DEFN
```

```
;; Consumes an s-expression and generates a DEFN.
(define (parseDefn sexp)
  (match sexp
    [`(,(? valid-identifier? id) ,expr) (defn id (parseExpr expr))
       ]
    [else (error 'parseDefn "Invalid␣definition:␣~s" sexp)]))

;; parseExprD : any -> EXPR-D
;; Consumes an s-expression and generates an EXPR-D.
(define (parseExprD sexp)
  (match sexp
    [(? valid-identifier?) (idD sexp)]
    [(? number?) (numD sexp)]
    [else (error 'parseExprD "Invalid␣expr:␣~s" sexp)]))

;; parseImage : any -> IMAGE
;; Consumes an s-expression and generates an IMAGE.
(define (parseImage sexp)
  (match sexp
    [`(circle ,expr1 ,str1 ,expr2 ,expr3) (circ (parseExprD expr1)
                                                str1
                                                (parseExprD expr2)
                                                (parseExprD expr3)
                                                   )]
    [else (error 'parseExprD "Invalid␣expr:␣~s" sexp)]))

;; parseProgramI : any -> PROGRAM-I
;; Consumes an s-expression and generates a PROGRAM-I.
(define (parseProgramI sexp)
  (match sexp
    [(list defns images worldClockRate tickSize)
     (programI (map parseDefn defns)
               (map parseImage images)
               worldClockRate
               tickSize)]))

;; parseProgramC : any -> PROGRAM-C
;; Consumes an s-expression and generates a PROGRAM-C.
(define (parseProgramC sexp)
  (local [(define (parseMapping pair)
            (mapping (first pair)
                     (second pair)))
          (define (parseValMapping pair)
            (val-mapping (first pair)
                         (num (second pair))))]
  (match sexp
```

```
    [(list defns ins outs vals)
     (programC (map parseDefn defns)
               (map parseMapping ins)
               (map parseMapping outs)
               (map parseValMapping vals))])))
```

Appendix D

Desugar.rkt

The following code is used to desugar Planck programs.

```
#lang racket

(provide (all-defined-out))

(require "Language.rkt")

(require data/queue)
(require plai/datatype)

;; desugarIf : List<DEFN> -> List<DEFN>
;; Desugars all instance of _if to the form {id {_if {id} {id} {id
   }}}
(define (desugarIf listOfDefn)
  (local [;; Count variable references for redefinitions.
          (define count (box 1))
          (define defnQueue (make-queue))

          ;; Symbol EXPR -> Symbol
          ;; Enqueues the given EXPR onto the defnQueue with a new
              id.
          ;; Returns the new id.
          (define (enqueueExpr id expr)
            (let* [(n (unbox count))
                   (newId (string->symbol (string-append "_"
                                                         (symbol->
                                                            string
                                                             id)
                                                         (number->
                                                            string
                                                             n))))
                                                           ]
              (begin (set-box! count (+ n 1))
                     (enqueue! defnQueue (defn newId expr))
                     newId)))

          ;; List<DEFN> -> List<DEFN>
```

```
;; Push definitions into queue and pass them on for
   processing.
(define (desugar_if loDefn)
  (begin (map (lambda (def) (enqueue! defnQueue def))
     loDefn)
         (processDefnQueue)))

;; Void -> List<DEFN>
;; Recursively process defnQueue
(define (processDefnQueue)
  (if (queue-empty? defnQueue)
      empty
      (cons (desugar (dequeue! defnQueue))
            (processDefnQueue))))

;; DEFN -> DEFN
;; Desugars _if recursively.
(define (desugar def)
  (local [(define id (defn-id def))
          (define expr (defn-expr def))
          (define (desugarH expr depth)
            (type-case EXPR expr
              [iden (_id) (iden _id)]
              [num (n) (num n)]
              [bool (b) (bool b)]
              [unaryNumOp (op uExpr)
                          (unaryNumOp op
                                      (desugarH uExpr (+
                                          1 depth)))]
              [binaryNumOp (op lhs rhs)
                           (binaryNumOp op
                                        (desugarH lhs (+
                                            1 depth))
                                        (desugarH rhs (+
                                            1 depth)))]
              [binaryLogOp (op lhs rhs)
                           (binaryLogOp op
                                        (desugarH lhs (+
                                            1 depth))
                                        (desugarH rhs (+
                                            1 depth)))]
              [integrate (iExpr) (integrate (desugarH
                 iExpr (+ 1 depth)))]
              [derivative (dExpr) (derivative (desugarH
                 dExpr (+ 1 depth)))]
              [_if (condExpr trueExpr falseExpr)
```

```
                          (if (= depth 0)
                              (_if (if (iden? condExpr) condExpr(
                                 iden (enqueueExpr id condExpr)))
                                   (if (iden? trueExpr) trueExpr (
                                      iden (enqueueExpr id
                                      trueExpr)))
                                   (if (iden? falseExpr) falseExpr
                                       (iden (enqueueExpr id
                                      falseExpr))))
                              (iden (enqueueExpr id (_if condExpr
                                 trueExpr falseExpr))))]
                        [passForward () (passForward)]))]
             (defn id (desugarH expr 0))))]
   (desugar_if listOfDefn)))

;; desugarIntegAndDiff : List<DEFN> -> List<DEFN>
;; Desugars all instances of integration to the form {<id> {
   integrate <id>}}.
(define (desugarIntegAndDiff listOfDefn)
  (local [;; Count variable references for redefinitions.
          (define count (box 1))
          (define defnQueue (make-queue))

          ;; Symbol EXPR -> Symbol
          ;; Enqueues the given EXPR onto the defnQueue with a new
              id.
          ;; Returns the new id.
          (define (enqueueExpr id expr)
            (let* [(n (unbox count))
                   (newId (string->symbol (string-append "_"
                                                         (symbol->
                                                            string
                                                             id)
                                                         (number->
                                                            string
                                                             n))))
                                                           ]
              (begin (set-box! count (+ n 1))
                     (enqueue! defnQueue (defn newId expr))
                     newId)))

          ;; List<DEFN> -> List<DEFN>
          ;; Push definitions into queue and pass them on for
             processing.
          (define (desugarIntegs loDefn)
```

```
  (begin (map (lambda (def) (enqueue! defnQueue def))
     loDefn)
         (processDefnQueue)))

;; Void -> List<DEFN>
;; Recursively process defnQueue
(define (processDefnQueue)
  (if (queue-empty? defnQueue)
      empty
      (cons (desugar (dequeue! defnQueue))
            (processDefnQueue))))

;; DEFN -> DEFN
;; Desugars integratations recursively.
(define (desugar def)
  (local [(define id (defn-id def))
          (define expr (defn-expr def))
          (define (desugarH expr depth)
            (type-case EXPR expr
              [iden (_id) (iden _id)]
              [num (n) (num n)]
              [bool (b) (bool b)]
              [unaryNumOp (op uExpr)
                          (unaryNumOp op
                                      (desugarH uExpr (+
                                         1 depth)))]
              [binaryNumOp (op lhs rhs)
                           (binaryNumOp op
                                        (desugarH lhs (+
                                            1 depth))
                                        (desugarH rhs (+
                                            1 depth)))]
              [binaryLogOp (op lhs rhs)
                           (binaryLogOp op
                                        (desugarH lhs (+
                                            1 depth))
                                        (desugarH rhs (+
                                            1 depth)))]
              [integrate (iExpr)
                         (cond [(and (= 0 depth) (iden?
                            iExpr))
                                (integrate iExpr)]
                               [(= 0 depth)
                                (integrate (iden (
                                   enqueueExpr id iExpr)
                                   ))]
```

```
                                          [else
                                             (iden (enqueueExpr id
                                                (integrate iExpr)))
                                                ])]
                              [derivative (dExpr)
                                          (cond [(and (= 0 depth) (iden?
                                              dExpr))
                                                 (derivative dExpr)]
                                                [(= 0 depth)
                                                 (derivative (iden (
                                                    enqueueExpr id dExpr)
                                                    ))]
                                                [else
                                                  (iden (enqueueExpr id
                                                     (derivative dExpr))
                                                     )])]
                              [_if (condExpr trueExpr falseExpr) expr]
                              [passForward () (passForward)]))]
                    (defn id (desugarH expr 0)))))]
     (desugarIntegs listOfDefn)))

;; desugar : List<DEFN> -> List<DEFN>
;; Apply desugarings
(define (desugar listOfDefn)
  (desugarIntegAndDiff (desugarIf listOfDefn)))

;; desugarProgramI : programI -> programI
;; Apply desugarings to an interpreted program
(define (desugarProgramI program)
  (programI (desugar (programI-loDefn program))
            (programI-loImage program)
            (programI-worldClockRate program)
            (programI-tickSize program)))
```

Appendix E

ExpressionStore.rkt

The following code implements the expression store for interpreted programs.

```
#lang racket

(provide (all-defined-out))

(require "Language.rkt")

(define exprStore? hash?)

;; createExprStore : -> ExprStore
;; Creates a new EXPR Store.
(define (createExprStore)
  (make-hash))

;; getExpr : ExprStore id -> EXPR
;; setExpr : ExprStore id EXPR -> ExprStore
;; Enter and retrieve values from an ExprStore respectively.
;; Note that setExpr throws an error if a identifier is redefined.
(define (getExpr store id)
  (hash-ref store id))

(define (setExpr store id val)
  (begin (if (hash-has-key? store id)
             (error 'setExpr "Duplicate␣value␣detected.")
             (hash-set! store id val))
         store))

;; populateExprStore : List<DEFN> -> ExprStore
;; Create an ExprStore from a list of definitions.
(define (populateExprStore listOfDefn)
  (local [(define (addExpr defn store)
            (setExpr store (defn-id defn) (defn-expr defn)))]
    (foldl addExpr (createExprStore) listOfDefn)))

;; exprStore->listIdens : ExprStore -> List<iden>
;; Returns a list of all of the ids in the key store.
(define (exprStore->listIdens exprStore)
  (hash-keys exprStore))
```

Appendix F

World.rkt

The following code implements the world for interpreted programs.

```
#lang racket

(provide (all-defined-out))

(require "Language.rkt")

(define world? hash?)

;; createWorld : -> WORLD
;; Creates a new world.
(define (createWorld)
  (make-hash))

;; getValW : WORLD id -> number
;; setValW : WORLD id number -> WORLD
;; Enter and retrieve values from the world respectively.
(define (getValW world id)
  (hash-ref world id))

(define (setValW world id val)
  (begin (hash-set! world id val)
         world))

;; inW : World id -> boolean
;; Determines whether an id is present in the world.
(define (inW world id)
  (hash-has-key? world id))
```

Appendix G

Preprocessing.rkt

The following code is used to preprocess Planck programs.

```
#lang racket

(provide (all-defined-out))

(require "Desugar.rkt")
(require "Language.rkt")

;; injectTime : List<Defn> -> List<Defn>
;; Add identifer for current time.
(define (injectTime loDefn)
  (cons (defn 't (integrate (num 1)))
        loDefn))

;; injectDeltaT : List<Defn> Number -> List<Defn>
;; Add an identifer for time step.
(define (injectDeltaT loDefn deltaVal)
  (cons (defn 'delta_t (num deltaVal))
        loDefn))

;; injectInputsI : List<Defn> -> List<Defn>
;; Add identifiers mapped to keyboard inputs.
(define (injectInputsI loDefn)
  (cons (defn 'ku (passForward))
        (cons (defn 'kd (passForward))
              (cons (defn 'kl (passForward))
                    (cons (defn 'kr (passForward))
                          loDefn)))))

;; preProcessProgramI : Program -> Program
;; Modify program before interpretation.
(define (preProcessProgramI program)
  (let [(loDefn (programI-loDefn program))
        (loImage (programI-loImage program))
        (worldClockRate (programI-worldClockRate program))
        (tickSize (programI-tickSize program))]
    (programI (desugar (injectDeltaT (injectTime (injectInputsI
       loDefn))
```

```
                                        tickSize))
              loImage
              worldClockRate
              tickSize)))

;; inMap->Defn : Mapping -> Defn
;; Produce a program definition from an input mapping.
(define (inMap->Defn inMap)
  (defn (mapping-id2 inMap) (passForward)))

;; injectInputMappings : List<Mapping> -> List<Defn>
(define (injectInputMappings inMappings loDefn)
  (foldl inMap->Defn loDefn inMappings))

;; preProcessProgramC : Program -> Program
;; Modify program before compilation.
(define (preProcessProgramC program)
  (let [(loDefn (programC-loDefn program))
        (inputMappings (programC-inputMappings program))
        (outputMappings (programC-outputMappings program))
        (deviceValues (programC-deviceValues program))]
    (programC (desugar (foldl cons
                              loDefn
                              (map inMap->Defn inputMappings)))
              inputMappings
              outputMappings
              deviceValues)))
```

Appendix H

Interpretation.rkt

The following code is used to interpret Planck programs.

```
#lang racket

(provide (all-defined-out))

(require "Desugar.rkt")
(require "ExprStore.rkt")
(require "Language.rkt")
(require "Parser.rkt")
(require "PreProcessing.rkt")
(require "World.rkt")

(require 2htdp/image)
(require 2htdp/universe)
(require plai/datatype)

(struct Stores (worldStore exprStore))

;; initInterp : EXPR World ExprStore -> Number
;; The first interpretation of the EXPR.
;; Detects cycles in definitions.
(define (initInterp expr world exprStore)
  (local [(define visitedSet (mutable-set))
          (define (initInterpHelper expr world)
            (type-case EXPR expr
              [iden (id) (cond [(inW world id) (getValW world id)]
                               [(set-member? visitedSet id) (error
                                   'initInterp "Cycle␣detected.")]
                               [else (begin (set-add! visitedSet
                                  id)
                                            (setValW world
                                                     id
                                                     (
                                                        initInterpHelper
                                                         (getExpr
                                                        exprStore
                                                        id)
```

```
                                                                    world
                                                                       )
                                                                       )

                                              (getValW world id))])]
                [num (n) n]
                [bool (b) b]
                [unaryNumOp (op uExpr)
                  (op (initInterpHelper uExpr world))]
                [binaryNumOp (op lhs rhs)
                  (op (initInterpHelper lhs world)
                      (initInterpHelper rhs world))]
                [binaryLogOp (op lhs rhs)
                  (op (initInterpHelper lhs world)
                      (initInterpHelper rhs world))]
                [integrate (n) 0]
                [derivative (n) 1]
                [_if (condExpr trueExpr falseExpr)
                     (let [(condVal (initInterpHelper condExpr world
                        ))]
                       (if (boolean? condVal)
                           (if condVal
                               (initInterpHelper trueExpr world)
                               (initInterpHelper falseExpr world))
                           (error 'initInterp "First␣expression␣in␣
                              an␣if␣statement␣must␣evaluate␣to␣
                              boolean␣value.")))]
                [passForward () 0]))]
    (initInterpHelper expr world)))

;; initializeStores : List<DEFN> -> Stores
;; Initializes the world and expression stores from a list of
   definitions.
(define (initializeStores listOfDefn)
  (let [(exprStore (populateExprStore listOfDefn))]
    (local [(define (initializeWorldHelper listOfDefn world)
              (if (empty? listOfDefn)
                  world
                  (let* [(defn (first listOfDefn))
                         (id (defn-id defn))
                         (expr (defn-expr defn))]
                    (initializeWorldHelper (rest listOfDefn)
                                           (setValW world
                                                    id
                                                    (initInterp expr
                                                        world
```

```
                                                    exprStore))))
                                                    ))]
    (Stores (initializeWorldHelper listOfDefn (createWorld))
            exprStore))))

;; updateWorld : World ExprStore -> World
;; Updates the world
(define (updateWorld oldWorld exprStore)
  (local [(define (updateWorld newWorld idSet)
            (if (set-empty? idSet)
                newWorld
                (let* [(id (set-first idSet))
                       (expr (getExpr exprStore id))]
                  (begin (set-remove! idSet id)
                         (updateWorld (updateInterpWithId expr id
                            newWorld idSet) idSet)))))

          (define (updateInterpWithId expr id newWorld idSet)
            (local [(define (updateInterp expr)
                      (type-case EXPR expr
                        [iden (exprId) (if (set-member? idSet
                           exprId)
                                           (begin (set-remove!
                                              idSet exprId)
                                                  (
                                                     updateInterpWithId
                                                      (getExpr
                                                     exprStore
                                                     exprId)
                                                                    exprId

                                                                    newWorld

                                                                    idSet
                                                                       )

                                                   (getValW
                                                      newWorld
                                                      exprId))
                                           (getValW newWorld
                                              exprId))]
                        [num (n) n]
                        [bool (b) b]
                        [unaryNumOp (op uExpr) (op (updateInterp
                           uExpr))]
```

```
                        [binaryNumOp (op lhs rhs) (op (
                           updateInterp lhs)
                                                      (
                                                         updateInterp
                                                          rhs))]
                        [binaryLogOp (op lhs rhs) (op (
                           updateInterp lhs)
                                                      (
                                                         updateInterp
                                                          rhs))]
                        [integrate (exprId) (+ (getValW oldWorld
                           id)
                                               (* (getValW
                                                  oldWorld '
                                                  delta_t)
                                                  (getValW
                                                     oldWorld (
                                                     iden-id
                                                     exprId))))]
                        [derivative (exprId) (/ (- (updateInterp
                           exprId)
                                                   (getValW
                                                      oldWorld (
                                                      iden-id
                                                      exprId)))
                                                (getValW oldWorld
                                                   'delta_t))]
                        [_if (condExpr trueExpr falseExpr)
                             (let [(condVal (updateInterp condExpr
                                ))]
                               (if (boolean? condVal)
                                   (if condVal
                                       (updateInterp trueExpr)
                                       (updateInterp falseExpr))
                                   (error 'initInterp "First␣
                                      expression␣in␣if␣statement␣
                                      must␣evaluate␣to␣boolean␣
                                      value.")))]
                        [passForward () (getValW oldWorld id)]))]
            (setValW newWorld id (updateInterp expr))))]
    (updateWorld (createWorld) (list->mutable-set (exprStore->
       listIdens exprStore)))))

;; interpExprD : EXPR-D World -> Number
(define (interpExprD expr world)
  (type-case EXPR-D expr
```

```
    [idD (id) (getValW world id)]
    [numD (n) n]))

;; interpImage : IMAGE World -> Image
(define (interpImages loImages world width height)
  (local [(define (generateImage image)
            (type-case IMAGE image
              [circ (rad color x y) (circle (interpExprD rad world
                 ) "solid" color)]))
          (define (placeImages image scene)
            (type-case IMAGE image
              [circ (rad color exprX exprY)
                    (place-image (generateImage image)
                                 (detX exprX world width)
                                 (detY exprY world height)
                                 scene)]))]
    (foldl placeImages (empty-scene width height) loImages)))


;; detX : id World width -> Number
;; Retrieves an X value from the world and converts it from
;; Cartesian coordinates to Racket coordinates based on the width.
(define (detX expr world width)
  (let [(val (interpExprD expr world))]
    (+ val (/ width 2))))

;; detY : id World width -> Number
;; Retrieves a Y value from the world and converts it from
;; Cartesian coordinates to Racket coordinates based on the height
   .
(define (detY expr world height)
  (let [(val (interpExprD expr world))]
    (- (/ height 2) val)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Run Programs

(define (run-program program)
  (let* ([prog (preProcessProgramI (parseProgramI program))]
         [stores (initializeStores (programI-loDefn prog))]
         [world (Stores-worldStore stores)]
         [exprStore (Stores-exprStore stores)]
         [imageExprs (programI-loImage prog)]
         [clockRate (programI-worldClockRate prog)])
    (big-bang world
              (on-tick (lambda (world)
                         (updateWorld world exprStore))
```

```
          clockRate)
(on-draw (lambda (world)
           (interpImages imageExprs world 1000 1000)
             ))
(on-key (lambda (world key)
          (cond [(key=? "up" key)    (setValW world
             'ku 1)]
                [(key=? "down" key)  (setValW world
                   'kd 1)]
                [(key=? "left" key)  (setValW world
                   'kl 1)]
                [(key=? "right" key) (setValW world
                   'kr 1)]
                [else world])))
(on-release (lambda (world key)
              (cond [(key=? "up" key)    (setValW
                 world 'ku 0)]
                    [(key=? "down" key)  (setValW
                       world 'kd 0)]
                    [(key=? "left" key)  (setValW
                       world 'kl 0)]
                    [(key=? "right" key) (setValW
                       world 'kr 0)]
                    [else world])))))))
```

Appendix I

CodeGeneration.rkt

The following code is used to generate Arduino C from Planck programs.

```
#lang racket

(provide (all-defined-out))

(require "Language.rkt")
(require "PreProcessing.rkt")

(require plai/datatype)

(define binaryLogOpMap (make-hash (list (cons '_and "&&")
                                        (cons '_or  "||"))))

;; CODE TEMPLATES
(define stateFunctionTemplate
  "double␣State_~a()␣{~nif␣(notUpdated[~a])␣{~ncurrWorld[~a]␣=␣~a
     ;~nnotUpdated[~a]␣=␣0;~n}~nreturn␣currWorld[~a];~n}")

(define stateFunctionCondTemplate
  "double␣State_~a()␣{~nif␣(notUpdated[~a])␣{~nif␣(State_~a())␣{~
     ncurrWorld[~a]␣=␣State_~a();~n}␣else␣{~ncurrWorld[~a]␣=␣
     State_~a();~n}~nnotUpdated[~a]␣=␣0;~n}~nreturn␣currWorld[~a
     ];~n}")

(define stateFunctionPassForwardTemplate
  "double␣State_~a()␣{~nreturn␣currWorld[~a];~n}")

(define initStateFunctionTemplate
  "double␣InitState_~a()␣{~nif␣(notUpdated[~a])␣{~ncurrWorld[~a]␣=
     ␣~a;~nnotUpdated[~a]␣=␣0;~n}~nreturn␣currWorld[~a];~n}")

(define initStateFunctionCondTemplate
  "double␣InitState_~a()␣{~nif␣(notUpdated[~a])␣{~nif␣(State_~a())
     ␣{~ncurrWorld[~a]␣=␣State_~a();~n}␣else␣{~ncurrWorld[~a]␣=␣
     State_~a();~n}~nnotUpdated[~a]␣=␣0;~n}~nreturn␣currWorld[~a
     ];~n}")

(define initStateFunctionPassForwardTemplate
```

```
  "double␣InitState_~a()␣{~nreturn␣currWorld[~a];~n}")

(define ifTemplate
  "if␣(~a)␣{~nreturn␣~a;}~n␣else␣{~nreturn␣~a;~n}")

(define unaryFunctionTemplate
  "~a(~a)")

(define inputMappingTemplate
  "currWorld[~a]␣=␣~a;\n")

(define initOutputMappingTemplate
  "~a␣=␣InitState_~a();\n"
)

(define outputMappingTemplate
  "~a␣=␣State_~a();\n"
)

(define deviceValueTemplate
  "~a␣=␣~a;\n")

;; CODE GENERATION FUNCTIONS
(define (generateSymbolTable loDefn)
  (local [(define count (box 0))
          (define (getCount)
            (let [(n (unbox count))]
              (begin (set-box! count (+ n 1))
                     n)))
          (define symbolTable (make-hash))
          (define (addToSymbolTable defn)
            (hash-set! symbolTable (defn-id defn) (getCount)))]
    (begin (map addToSymbolTable loDefn)
           symbolTable)))

(define (generateDefines symbolTable)
  (local [(define (generateDefinesHelper keyValPair)
            (let [(key (car keyValPair))
                  (val (cdr keyValPair))]
            (format "#define␣~a␣~a~n" key val)))
          (define numOfStates (hash-count symbolTable))]
    (foldl string-append (format "#define␣NumOfStates␣~a"
       numOfStates)
                         (map generateDefinesHelper (hash->list
                            symbolTable)))))
```

```
(define (generateInitStateFunctionProtos loDefn)
  (local [(define (prototypeGenerator defn)
            (format "double␣InitState_~a();~n" (defn-id defn)))]
    (foldl string-append "" (map prototypeGenerator loDefn))))

(define (generateStateFunctionProtos loDefn)
  (local [(define (prototypeGenerator defn)
            (format "double␣State_~a();~n" (defn-id defn)))]
    (foldl string-append "" (map prototypeGenerator loDefn))))

(define (generateInitFunctionCalls loDefn)
  (local [(define (callGenerator defn)
            (format "InitState_~a();~n" (defn-id defn)))]
    (foldl string-append "" (map callGenerator loDefn))))

(define (generateFunctionCalls loDefn)
  (local [(define (callGenerator defn)
            (format "State_~a();~n" (defn-id defn)))]
    (foldl string-append "" (map callGenerator loDefn))))

(define (generateInputMappings inputMappings)
  (local [(define (inputMapGenerator inMap)
            (format inputMappingTemplate
                    (mapping-id2 inMap)
                    (mapping-id1 inMap)))]
    (foldl string-append "" (map inputMapGenerator inputMappings))
      ))

(define (generateInitOutputMappings outputMappings)
  (local [(define (outputMapGenerator outMap)
            (format initOutputMappingTemplate
                    (mapping-id2 outMap)
                    (mapping-id1 outMap)))]
    (foldl string-append "" (map outputMapGenerator outputMappings
      ))))

(define (generateOutputMappings outputMappings)
  (local [(define (outputMapGenerator outMap)
            (format outputMappingTemplate
                    (mapping-id2 outMap)
                    (mapping-id1 outMap)))]
    (foldl string-append "" (map outputMapGenerator outputMappings
      ))))

(define (generateDeviceValueAssignments deviceValues)
  (local [(define (deviceValueGenerator devVal)
```

```
            (format deviceValueTemplate
                    (val-mapping-id devVal)
                    (num-n (val-mapping-val devVal))))]
    (foldl string-append "" (map deviceValueGenerator deviceValues
       ))))

(define (generateInitStateFunctions loDefn)
  (local [(define (functionBodyGenerator stateId expr)
            (type-case EXPR expr
              [iden (id) (format "InitState_~a()" id)]
              [num (n) n]
              [bool (b) (if b 'true 'false)]
              [unaryNumOp (op expr)
                (format unaryFunctionTemplate
                        (object-name op)
                        (functionBodyGenerator stateId expr))]
              [binaryNumOp (op lhs rhs)
                (format "(~a␣~a␣~a)"  (functionBodyGenerator
                   stateId lhs)
                                      (object-name op)
                                      (functionBodyGenerator
                                         stateId rhs))]
              [binaryLogOp (op lhs rhs)
                (format "(~a␣~a␣~a)" (functionBodyGenerator
                   stateId lhs)
                                     (hash-ref binaryLogOpMap
                                               (object-name op)
                                               (object-name op))
                                     (functionBodyGenerator
                                        stateId rhs))]
              [integrate (expr) "0"]
              [derivative (expr) "1"]
              [_if (condExpr trueExpr falseExpr)
                (error 'generatateInitStateFunctions
                       "_if␣should␣have␣been␣handled␣before␣this␣
                          point.")]
              [passForward ()
                (error 'generatateInitStateFunctions
                       "passForward␣should␣have␣been␣handled␣
                          before␣this␣point.")]))
          (define (stateFunctionGenerator defn)
            (let ([id (defn-id defn)]
                  [expr (defn-expr defn)])
              (type-case EXPR expr
                [_if (condExpr trueExpr falseExpr)
                  (let ([condId (iden-id condExpr)]
```

```
                        [trueId (iden-id trueExpr)]
                        [falseId (iden-id falseExpr)])
                    (format initStateFunctionCondTemplate id id
                                                        condId
                                                        id trueId
                                                        id falseId
                                                        id id))]
              [passForward ()
                (format initStateFunctionPassForwardTemplate id
                   id)]
              [else
                (format initStateFunctionTemplate  id id id
                                                (functionBodyGenerator
                                                    id expr)
                                                id id)]))))]
    (foldl string-append "" (map stateFunctionGenerator loDefn))))

(define (generateStateFunctions loDefn)
  (local [(define (functionBodyGenerator stateId expr)
            (type-case EXPR expr
              [iden (id) (format "State_~a()" id)]
              [num (n) n]
              [bool (b) (if b 'true 'false)]
              [unaryNumOp (op expr)
                (format unaryFunctionTemplate
                        (object-name op)
                        (functionBodyGenerator stateId expr))]
              [binaryNumOp (op lhs rhs)
                (format "(~a␣~a␣~a)"  (functionBodyGenerator
                   stateId lhs)
                                       (object-name op)
                                       (functionBodyGenerator
                                          stateId rhs))]
              [binaryLogOp (op lhs rhs)
                (format "(~a␣~a␣~a)" (functionBodyGenerator
                   stateId lhs)
                                     (hash-ref binaryLogOpMap
                                               (object-name op)
                                               (object-name op))
                                     (functionBodyGenerator
                                        stateId rhs))]
              [integrate (expr)
                (format "integrate(~a,␣~a)" stateId (iden-id expr)
                   )]
              [derivative (expr)
                (format "derivative(~a)" (iden-id expr))]
```

```
            [_if (condExpr trueExpr falseExpr)
              (error 'generatateStateFunctions
                     "_if␣should␣have␣been␣handled␣before␣this␣
                        point.")]
            [passForward ()
              (error 'generatateStateFunctions
                     "passForward␣should␣have␣been␣handled␣
                        before␣this␣point.")]))
        (define (stateFunctionGenerator defn)
          (let ([id (defn-id defn)]
                [expr (defn-expr defn)])
            (type-case EXPR expr
              [_if (condExpr trueExpr falseExpr)
                (let ([condId (iden-id condExpr)]
                      [trueId (iden-id trueExpr)]
                      [falseId (iden-id falseExpr)])
                  (format stateFunctionCondTemplate id id
                                                    condId
                                                    id trueId
                                                    id falseId
                                                    id id))]
              [passForward ()
                (format stateFunctionPassForwardTemplate id id)]
              [else
                (format stateFunctionTemplate  id id id
                                               (functionBodyGenerator
                                                   id expr)
                                               id id)]))))]
    (foldl string-append "" (map stateFunctionGenerator loDefn))))

(define (generateCode program templateFile)
  (let* [(op (open-input-file templateFile))
         (codeTemplate (port->string op))
         (preProgram (preProcessProgramC program))
         (inMap (programC-inputMappings preProgram))
         (outMap (programC-outputMappings preProgram))
         (deviceVals (programC-deviceValues preProgram))
         (loDefn (programC-loDefn preProgram))
         (SYMBOL_TABLE (generateSymbolTable loDefn))
         (DEFINES (generateDefines SYMBOL_TABLE))
         (INIT_STATE_FUNCTION_PROTOS (
            generateInitStateFunctionProtos loDefn))
         (STATE_FUNCTION_PROTOS (generateStateFunctionProtos
            loDefn))
         (INIT_STATE_FUNCTIONS (generateInitStateFunctions loDefn)
            )
```

```
      (STATE_FUNCTIONS (generateStateFunctions loDefn))
      (DEVICE_VALS (generateDeviceValueAssignments deviceVals))
      (INPUT_MAPPINGS (generateInputMappings inMap))
      (INIT_OUTPUT_MAPPINGS (generateInitOutputMappings outMap)
         )
      (OUTPUT_MAPPINGS (generateOutputMappings outMap))
      (FUNCTION_CALLS (generateFunctionCalls loDefn))
      (INIT_FUNCTION_CALLS (generateInitFunctionCalls loDefn))
      (INIT_SETUP (string-append INPUT_MAPPINGS
                                 INIT_FUNCTION_CALLS
                                 INIT_OUTPUT_MAPPINGS))
      (SETUP (string-append INPUT_MAPPINGS
                            FUNCTION_CALLS
                            OUTPUT_MAPPINGS))]
(begin (close-input-port op)
       (format codeTemplate DEFINES
                            INIT_STATE_FUNCTION_PROTOS
                            STATE_FUNCTION_PROTOS
                            INIT_STATE_FUNCTIONS
                            STATE_FUNCTIONS
                            DEVICE_VALS
                            INIT_SETUP
                            SETUP))))
```

Appendix J

Planck.rkt

The *Planck* module provides access to code interpretation and compilation through *interpretProgram* and *compileProgram* functions.

```
#lang racket

(provide (all-defined-out))

(require "Parser.rkt")
(require "Interpretation.rkt")
(require "CodeGeneration.rkt")

(define (interpretProgram program)
  (run-program program))

(define (compileProgram program templateFilePath outFilePath)
  (let [(progC (parseProgramC program))
        (op (open-output-file outFilePath
                              #:mode 'text #:exists 'replace))]
  (begin (write-string (generateCode progC templateFilePath)
                       op)
         (close-output-port op))))
```

Appendix K

General Template Format

The full format of a general template follows.

```
// IMPORTS
#include <arduino.h>

// DEFINES
~a

// TIME
double globalTime;
double prevTime;
double deltaT;

// WORLD
double worldStore1[NumOfStates] = {0};
double worldStore2[NumOfStates] = {0};
bool notUpdated[NumOfStates] = {0};
double *prevWorld;
double *currWorld;

// INTEGRATION AND DIFFERENTIATION
double integrate(int variableKey, int integrandKey) {
    double oldVariableValue = prevWorld[variableKey];
    double oldIntegrandValue = prevWorld[integrandKey];
    return oldVariableValue + deltaT * oldIntegrandValue;
}

double derivative(int derivativeKey) {
    return  (currWorld[derivativeKey] - prevWorld[derivativeKey])
       / deltaT;
}

// PROTOTYPES
~a
~a

// STATE INITIALIZATION FUNCTIONS
~a
```

```
// STATE FUNCTIONS
~a

// ----- Device Specific Code Starts -----
// Declare parameters for output mappings.
  double deviceOutputValue; // Example
// Helper functions for handling device I/O
// Special device parameters;
  double calibrationParam; // Example
// ----- Device Specific Code Ends   -----

void setup() {
    // Initialize Device Values
    ~a

    globalTime = 0;
    prevTime = micros();
    prevWorld = worldStore1;
    currWorld = worldStore2;
    memset(notUpdated, 1, NumOfStates * sizeof(bool));

    // ----- Device Specific Code Starts -----
    // Read sensor inputs into device input identifiers.
    // ----- Device Specific Code Ends   -----

    // Assign device input identifiers to program input
       identifiers.
    // Call all InitState functions.
    // Assign program output identifiers to device output
       identifiers.
    ~a

    // ----- Device Specific Code Starts -----
    // Handle device outputs.
    // ----- Device Specific Code Ends   -----
}

void loop() {
    double tNow = micros();
    deltaT = tNow - prevTime;
    prevTime = tNow;
    globalTime += deltaT;
    double* temp = prevWorld;
    prevWorld = currWorld;
    currWorld = temp;
    memset(notUpdated, 1, NumOfStates * sizeof(bool));
```

```
    // ----- Device Specific Code Starts -----
    // Read sensor inputs into device sensor identifiers.
    // ----- Device Specific Code Ends   -----

    // Assign device input identifiers to program input
       identifiers.
    // Call all State functions.
    // Assign program output identifiers to device output
       identifiers.
    ~a
    // ----- Device Specific Code Starts -----
    // Handle device outputs.
    // ----- Device Specific Code Ends   -----
}
```

Appendix L

Hapkit Template

The full Hapkit template follows.

```
// IMPORTS
#include <arduino.h>

// DEFINES
~a

// TIME
double globalTime;
double prevTime;
double deltaT;

// WORLD
double worldStore1[NumOfStates] = {0};
double worldStore2[NumOfStates] = {0};
bool notUpdated[NumOfStates] = {0};
double *prevWorld;
double *currWorld;

// INTEGRATION AND DIFFERENTIATION
double integrate(int variableKey, int integrandKey) {
    double oldVariableValue = prevWorld[variableKey];
    double oldIntegrandValue = prevWorld[integrandKey];
    return oldVariableValue + deltaT * oldIntegrandValue;
}

double derivative(int derivativeKey) {
    return  (currWorld[derivativeKey] - prevWorld[derivativeKey])
       / deltaT;
}

// PROTOTYPES
~a
~a

// STATE INITIALIZATION FUNCTIONS
~a
```

```
// STATE FUNCTIONS
~a

// ----- HapKit Specific Code Starts -----
// Pin declares
int pwmPin = 5;          // PWM output pin for motor 1
int dirPin = 8;          // direction output pin for motor 1
int sensorPosPin = A2;   // input pin for MR sensor

// Position tracking variables
int updatedPos = 0;      // keeps track of the latest updated value
    of the MR sensor reading
int rawPos = 0;          // current raw reading from MR sensor
int lastRawPos = 0;      // last raw reading from MR sensor
int lastLastRawPos = 0; // last last raw reading from MR sensor
int flipNumber = 0;      // keeps track of the number of flips over
    the 180deg mark
int tempOffset = 0;
int rawDiff = 0;
int lastRawDiff = 0;
int rawOffset = 0;
int lastRawOffset = 0;
const int flipThresh = 700;  // threshold to determine whether or
   not a flip over the 180 degree mark occurred
boolean flipped = false;

// Kinematics variables
double xh = 0;               // position of the handle [m]

// Force output variables
double force;            // force at the handle
double Tp = 0;               // torque of the motor pulley
double duty = 0;             // duty cycle (between 0 and 255)
unsigned int output = 0;     // output command to the motor

// Device Variables   (!!! From Measurements !!!)
double rPulley; //[m]
double rSector; //[m]
double rHandle; //[m]
// Fit: theta = m * count + theta0 (!!! From Calibration !!!)
double m = 0.0002;    //[rads / count]
double theta0 = -0.1403;  //[rads]

// Constants
const float Pi = 3.141593;
```

```
void setPwmFrequency(int pin, int divisor) {
  byte mode;
  if(pin == 5 || pin == 6 || pin == 9 || pin == 10) {
    switch(divisor) {
      case 1: mode = 0x01; break;
      case 8: mode = 0x02; break;
      case 64: mode = 0x03; break;
      case 256: mode = 0x04; break;
      case 1024: mode = 0x05; break;
      default: return;
    }
    if(pin == 5 || pin == 6) {
      TCCR0B = TCCR0B & 0b11111000 | mode;
    } else {
      TCCR1B = TCCR1B & 0b11111000 | mode;
    }
  } else if(pin == 3 || pin == 11) {
    switch(divisor) {
      case 1: mode = 0x01; break;
      case 8: mode = 0x02; break;
      case 32: mode = 0x03; break;
      case 64: mode = 0x04; break;
      case 128: mode = 0x05; break;
      case 256: mode = 0x06; break;
      case 1024: mode = 0x7; break;
      default: return;
    }
    TCCR2B = TCCR2B & 0b11111000 | mode;
  }
}
// ----- HapKit Specific Code Ends -----

void setup() {
  // Initialize Device Values
  ~a

  globalTime = 0;
  prevTime = micros();
  prevWorld = worldStore1;
  currWorld = worldStore2;
  memset(notUpdated, 1, NumOfStates * sizeof(bool));

  // ----- HapKit Specific Code Starts -----
  // Set up serial communication
  Serial.begin(9600);
  // Set PWM frequency
```

```
  setPwmFrequency(pwmPin,1);
  // Input pin
  pinMode(sensorPosPin, INPUT); // set MR sensor pin to be an
     input
  // Output pins
  pinMode(pwmPin, OUTPUT);  // PWM pin for motor A
  pinMode(dirPin, OUTPUT);  // dir pin for motor A
  // Initialize motor
  analogWrite(pwmPin, 0);     // set to not be spinning (0/255)
  digitalWrite(dirPin, LOW);  // set direction
  // Initialize position valiables
  lastLastRawPos = analogRead(sensorPosPin);
  lastRawPos = analogRead(sensorPosPin);
  // ----- HapKit Specific Code Ends -----

  // Assign device input identifiers to program input identifiers.
  // Call all InitState functions.
  // Assign program output identifiers to device output
     identifiers.
  ~a
}

void loop() {
  double tNow = micros();
  deltaT = tNow - prevTime;
  prevTime = tNow;
  globalTime += deltaT;
  double* temp = prevWorld;
  prevWorld = currWorld;
  currWorld = temp;
  memset(notUpdated, 1, NumOfStates * sizeof(bool));

  // ----- HapKit Specific Code Starts -----
  // Get voltage output by MR sensor
  rawPos = analogRead(sensorPosPin);  //current raw position from
     MR sensor
  // Calculate differences between subsequent MR sensor readings
  rawDiff = rawPos - lastRawPos;          //difference btwn
     current raw position and last raw position
  lastRawDiff = rawPos - lastLastRawPos;  //difference btwn
     current raw position and last last raw position
  rawOffset = abs(rawDiff);
  lastRawOffset = abs(lastRawDiff);

  // Update position record-keeping variables
  lastLastRawPos = lastRawPos;
```

```
lastRawPos = rawPos;

// Keep track of flips over 180 degrees
if ((lastRawOffset > flipThresh) && (!flipped)) { // enter this
   anytime the last offset is greater than the flip threshold
   AND it has not just flipped
  if (lastRawDiff > 0) {  // check to see which direction the
     drive wheel was turning
    flipNumber--;        // cw rotation
  } else {                 // if(rawDiff < 0)
    flipNumber++;        // ccw rotation
  }

  if (rawOffset > flipThresh) { // check to see if the data was
     good and the most current offset is above the threshold
    updatedPos = rawPos + flipNumber*rawOffset; // update the
       pos value to account for flips over 180deg using the most
        current offset
    tempOffset = rawOffset;
  } else {                     // in this case there was a blip
     in the data and we want to use lastactualOffset instead
    updatedPos = rawPos + flipNumber*lastRawOffset;  // update
       the pos value to account for any flips over 180deg using
       the LAST offset
    tempOffset = lastRawOffset;
  }
  flipped = true;             // set boolean so that the next
     time through the loop won't trigger a flip
} else {                        // anytime no flip has occurred
  updatedPos = rawPos + flipNumber*tempOffset; // need to update
      pos based on what most recent offset is
  flipped = false;
}

double ts = updatedPos * m + theta0;  //[rads]
xh = ts * (rHandle*rPulley) / rSector;
// ----- HapKit Specific Code Ends -----

// Assign device input identifiers to program input identifiers.
// Call all State functions.
// Assign program output identifiers to device output
   identifiers.
~a

// ----- HapKit Specific Code Starts -----
```

```
  // Compute the require motor pulley torque (Tp) to generate that
      force
  Tp = (force * rHandle * rPulley) / rSector;

  // Determine correct direction for motor torque
  if(force < 0) {
    digitalWrite(dirPin, HIGH);
  } else {
    digitalWrite(dirPin, LOW);
  }

  // Compute the duty cycle required to generate Tp (torque at the
      motor pulley)
  duty = sqrt(abs(Tp)/0.0125); // the constant is 0.0125 for the
     Mabuchi motor and 0.03 for the Maxon A-max motor)

  // Make sure the duty cycle is between 0 and 100%
  if (duty > 1) {
    duty = 1;
  } else if (duty < 0) {
    duty = 0;
  }

  output = (int)(duty* 255);   // convert duty cycle to output
     signal
  analogWrite(pwmPin,output);  // output the signal
  // ----- HapKit Specific Code Ends -----
}
```

Appendix M

Compiled Spring Program

The code generated for the spring program in Figure 3.7 follows.

```
// IMPORTS
#include <arduino.h>

// DEFINES
#define t 0
#define F 3
#define k 2
#define x 1
#define NumOfStates 4

// TIME
double globalTime;
double prevTime;
double deltaT;

// WORLD
double worldStore1[NumOfStates] = {0};
double worldStore2[NumOfStates] = {0};
bool notUpdated[NumOfStates] = {0};
double *prevWorld;
double *currWorld;

// INTEGRATION AND DIFFERENTIATION
double integrate(int variableKey, int integrandKey) {
    double oldVariableValue = prevWorld[variableKey];
    double oldIntegrandValue = prevWorld[integrandKey];
    return oldVariableValue + deltaT * oldIntegrandValue;
}

double derivative(int derivativeKey) {
    return  (currWorld[derivativeKey] - prevWorld[derivativeKey])
       / deltaT;
}

// PROTOTYPES
double InitState_F();
double InitState_k();
```

```
double InitState_x();
double InitState_t();

double State_F();
double State_k();
double State_x();
double State_t();

// STATE INITIALIZATION FUNCTIONS
double InitState_F() {
  if (notUpdated[F]) {
    currWorld[F] = -((InitState_k() * InitState_x()));
    notUpdated[F]=0;
  }
  return currWorld[F];
}

double InitState_k() {
  if (notUpdated[k]) {
    currWorld[k] = 100;
    notUpdated[k] = 0;
  }
  return currWorld[k];
}

double InitState_x() {
  return currWorld[x];
}

double InitState_t() {
  return currWorld[t];
}

// STATE FUNCTIONS
double State_F() {
  if (notUpdated[F]) {
    currWorld[F] = -((State_k() * State_x()));
    notUpdated[F] = 0;
  }
  return currWorld[F];
}

double State_k() {
  if (notUpdated[k]) {
    currWorld[k] = 100;
    notUpdated[k] = 0;
```

```
  }
  return currWorld[k];
}

double State_x() {
  return currWorld[x];
}

double State_t() {
  return currWorld[t];
}

// ----- HapKit Specific Code Starts -----
// Pin declares
int pwmPin = 5;          // PWM output pin for motor 1
int dirPin = 8;          // direction output pin for motor 1
int sensorPosPin = A2;   // input pin for MR sensor

// Position tracking variables
int updatedPos = 0;      // keeps track of the latest updated value
    of the MR sensor reading
int rawPos = 0;          // current raw reading from MR sensor
int lastRawPos = 0;      // last raw reading from MR sensor
int lastLastRawPos = 0; // last last raw reading from MR sensor
int flipNumber = 0;      // keeps track of the number of flips over
    the 180deg mark
int tempOffset = 0;
int rawDiff = 0;
int lastRawDiff = 0;
int rawOffset = 0;
int lastRawOffset = 0;
const int flipThresh = 700;  // threshold to determine whether or
   not a flip over the 180 degree mark occurred
boolean flipped = false;

// Kinematics variables
double xh = 0;               // position of the handle [m]

// Force output variables
double force;            // force at the handle
double Tp = 0;               // torque of the motor pulley
double duty = 0;             // duty cycle (between 0 and 255)
unsigned int output = 0;     // output command to the motor

// Device Variables   (!!! From Measurements !!!)
double rPulley; //[m]
```

```
double rSector; //[m]
double rHandle; //[m]
// Fit: theta = m * count + theta0 (!!! From Calibration !!!)
double m = 0.0002;     //[rads / count]
double theta0 = -0.1403;  //[rads]

// Constants
const float Pi = 3.141593;

void setPwmFrequency(int pin, int divisor) {
  byte mode;
  if(pin == 5 || pin == 6 || pin == 9 || pin == 10) {
    switch(divisor) {
      case 1: mode = 0x01; break;
      case 8: mode = 0x02; break;
      case 64: mode = 0x03; break;
      case 256: mode = 0x04; break;
      case 1024: mode = 0x05; break;
      default: return;
    }
    if(pin == 5 || pin == 6) {
      TCCR0B = TCCR0B & 0b11111000 | mode;
    } else {
      TCCR1B = TCCR1B & 0b11111000 | mode;
    }
  } else if(pin == 3 || pin == 11) {
    switch(divisor) {
      case 1: mode = 0x01; break;
      case 8: mode = 0x02; break;
      case 32: mode = 0x03; break;
      case 64: mode = 0x04; break;
      case 128: mode = 0x05; break;
      case 256: mode = 0x06; break;
      case 1024: mode = 0x7; break;
      default: return;
    }
    TCCR2B = TCCR2B & 0b11111000 | mode;
  }
}
// ----- HapKit Specific Code Ends -----

void setup() {
  // Initialize Device Values
  theta0 = -0.1403;
  m = 0.0002;
  rHandle = 0.074;
```

```
  rSector = 0.077;
  rPulley = 0.005;

  globalTime = 0;
  prevTime = micros();
  prevWorld = worldStore1;
  currWorld = worldStore2;
  memset(notUpdated, 1, NumOfStates * sizeof(bool));

  // ----- HapKit Specific Code Starts -----
  // Set up serial communication
  Serial.begin(9600);
  // Set PWM frequency
  setPwmFrequency(pwmPin,1);
  // Input pin
  pinMode(sensorPosPin, INPUT); // set MR sensor pin to be an
     input
  // Output pins
  pinMode(pwmPin, OUTPUT);  // PWM pin for motor A
  pinMode(dirPin, OUTPUT);  // dir pin for motor A
  // Initialize motor
  analogWrite(pwmPin, 0);     // set to not be spinning (0/255)
  digitalWrite(dirPin, LOW);  // set direction
  // Initialize position valiables
  lastLastRawPos = analogRead(sensorPosPin);
  lastRawPos = analogRead(sensorPosPin);
  // ----- HapKit Specific Code Ends -----

  // Assign device input identifiers to program input identifiers.
  // Call all InitState functions.
  // Assign program output identifiers to device output
     identifiers.
  currWorld[t] = globalTime;
  currWorld[x] = xh;
  InitState_F();
  InitState_k();
  InitState_x();
  InitState_t();
  force = InitState_F();
}

void loop() {
  double tNow = micros();
  deltaT = tNow - prevTime;
  prevTime = tNow;
  globalTime += deltaT;
```

```
double* temp = prevWorld;
prevWorld = currWorld;
currWorld = temp;
memset(notUpdated, 1, NumOfStates * sizeof(bool));

// ----- HapKit Specific Code Starts -----
// Get voltage output by MR sensor
rawPos = analogRead(sensorPosPin);  //current raw position from
   MR sensor
// Calculate differences between subsequent MR sensor readings
rawDiff = rawPos - lastRawPos;          //difference btwn
   current raw position and last raw position
lastRawDiff = rawPos - lastLastRawPos;  //difference btwn
   current raw position and last last raw position
rawOffset = abs(rawDiff);
lastRawOffset = abs(lastRawDiff);

// Update position record-keeping variables
lastLastRawPos = lastRawPos;
lastRawPos = rawPos;

// Keep track of flips over 180 degrees
if ((lastRawOffset > flipThresh) && (!flipped)) { // enter this
   anytime the last offset is greater than the flip threshold
   AND it has not just flipped
  if (lastRawDiff > 0) {  // check to see which direction the
     drive wheel was turning
    flipNumber--;        // cw rotation
  } else {                 // if(rawDiff < 0)
    flipNumber++;        // ccw rotation
  }

  if (rawOffset > flipThresh) { // check to see if the data was
     good and the most current offset is above the threshold
    updatedPos = rawPos + flipNumber*rawOffset; // update the
       pos value to account for flips over 180deg using the most
        current offset
    tempOffset = rawOffset;
  } else {                      // in this case there was a blip
     in the data and we want to use lastactualOffset instead
    updatedPos = rawPos + flipNumber*lastRawOffset;  // update
       the pos value to account for any flips over 180deg using
       the LAST offset
    tempOffset = lastRawOffset;
  }
```

```
  flipped = true;            // set boolean so that the next
     time through the loop won't trigger a flip
} else {                          // anytime no flip has occurred
  updatedPos = rawPos + flipNumber*tempOffset; // need to update
      pos based on what most recent offset is
  flipped = false;
}

double ts = updatedPos * m + theta0;  //[rads]
xh = ts * (rHandle*rPulley) / rSector;
// ----- HapKit Specific Code Ends -----

// Assign device input identifiers to program input identifiers.
// Call all State functions.
// Assign program output identifiers to device output
   identifiers.
currWorld[t] = globalTime;
currWorld[x] = xh;
State_F();
State_k();
State_x();
State_t();
force = State_F();

// ----- HapKit Specific Code Starts -----
// Compute the require motor pulley torque (Tp) to generate that
    force
Tp = (force * rHandle * rPulley) / rSector;

// Determine correct direction for motor torque
if(force < 0) {
  digitalWrite(dirPin, HIGH);
} else {
  digitalWrite(dirPin, LOW);
}

// Compute the duty cycle required to generate Tp (torque at the
    motor pulley)
duty = sqrt(abs(Tp)/0.0125); // the constant is 0.0125 for the
   Mabuchi motor and 0.03 for the Maxon A-max motor)

// Make sure the duty cycle is between 0 and 100%
if (duty > 1) {
  duty = 1;
} else if (duty < 0) {
  duty = 0;
```

```
  }

  output = (int)(duty* 255);   // convert duty cycle to output
     signal
  analogWrite(pwmPin,output);  // output the signal
  // ----- HapKit Specific Code Ends -----
}
```